MATLAB 5
Class Notes

-O-O-O-O-O-

Richard I. Shrager
Mathematical and Statistical
Consulting Laboratory
Center for Information Technology
National Institutes of Health
Bethesda MD 20892
Telephone: (301)496-1135
Fax: (301)402-4544
E-mail: ris@alw.nih.gov

-O-O-O-O-O-

April 29, 2000

Preface

This book is intended to serve as class notes for a two or three day course. The course itself serves as a quick scan of the material, after which the student may wish to examine the notes more closely. The emphasis is on the core language of MATLAB, that is, data types, program elements, and the syntax for using them. Peripheral (though very important) issues like input/output, graphics, and toolboxes, are not covered to any significant extent. Those topics tend to require long lists of options which are awkward to fit into a brief course, and which are thoroughly explained in the references below, or in specialized toolbox references (not listed). In contrast, these notes have much to say about indexing, which is the method for accessing parts of arrays. Every built-in data type in MATLAB is an array, placing indexing at the heart of the language. Often, the topics covered are approached from a different point of view than the references, with two goals in mind: 1) avoiding initial confusions that previous students have encountered in their early use of MATLAB, and 2) highlighting especially useful techniques from the many found in the references.

References

- The MathWorks, Inc. (1996) The MATLAB 5 Manuals, particularly those subtitled: "Using MATLAB" and "Using MATLAB Graphics".
- Duane Hanselman and Bruce Littlefield (1998) Mastering MATLAB 5: A Comprehensive Tutorial and Reference. Prentice Hall, Upper Saddle River NJ 07458.

Contents

Chapter 1

About matrices

1.1 Motivation

A matrix is a set of numbers arranged in a rectangular array. Matrices may be defined as follows in MATLAB:

```
>>a = [  1,  2,  3;
         4,  5,  6;
         7,  8,  9 ];
>>b = [ 10, 20, 30;
        40, 50, 60;
        70, 80, 90 ];
```

where >> is the MATLAB prompt. The elements (individual entries) in a row are separated by commas or blanks, and rows are separated by semicolons or by new lines. Matrices are given names in the same manner as scalars in ordinary algebra, and there is a matrix algebra which has a considerable resemblance to the algebra of scalars. In science, and in other fields as well, one may be faced with problems concerning large collections of similar numbers, e.g. ages, heights, weights, and blood pressures of many patients, or absorbance spectra measured at hundreds of wavelengths taken repeatedly over the time course of some reaction. Over the years, experience (accelerated by the advent of available computing power) has shown that there are certain common operations that one wishes to use on matrices. A notation has developed to accomodate these operations, with the benefit that one can now condense many scalar operations into very few matrix operations. But there is a cost for this convenience. In order to communicate ideas in such a notation, whole communities of people must learn the notation, communities that might otherwise prefer to avoid as much math as possible. So the benefits had better be worth the effort.

1.2 MATLAB-related projects

Because of the intense interest processing matrices, MATLAB is not the only matrix-oriented language. There are a few web sites that survey such languages. These are the ones I have visited.

1. `http://www.cen.uiuc.edu/~tawarmal/linux_old.html#matlab`
2. `http://www-ocean.tamu.edu/~baum/graphics-analysis.html`
3. `http://sal.kachinatech.com`

(Sorry, no hot links. These are paper notes after all.) At these sites you will find links to such MATLAB-like languages as SciLab and Octave, plus MATCOM, a facility for translating MATLAB code into C++. However, while MATLAB is pricey, it remains king of the hill, and at NIH we are fortunate to have facilities like Helix and ALW that have MATLAB licences available to its users.

1.3 Elementary matrix operations

For initial orientation in matrix operations, it is instructive to look at the resemblance between scalar and matrix operations. The operators `+` and `-` are essentially identical in both contexts, being defined only when the matrices are of the same size. The operation `c = a+b` means every `c(i,j) = a(i,j)+b(i,j)`, where `a(i,j)` is the element (scalar entry) of `a` in the ith row, jth column, and similarly for `b(i,j)` in `b` and `c(i,j)` in `c`. Subtraction works the same way. The sum or difference matrix `c` is the same size as `a` and `b`. Using `a` and `b` defined above, we get:

```
>>c = a + b              % When you do not
                         % end a statement
c = 11  22  33           % with a semicolon
    44  55  66           % the result(s)
    77  88  99           % of the statement
                         % is (are) displayed.
>>d = b - a

d =  9  18  27
    36  45  54
    63  72  81
```

The matrix literature tends to denote matrices with upper case letters, while the elements of those matrices are denoted by the corresponding subscripted lower case letters, e.g., the matrix M has elements $m_{i,j}$. MATLAB has a different convention. The symbol for the whole matrix is the same as the symbol for its elements, and an element has its indices suffixed in parentheses rather than subscripted, Thus in MATLAB, the matrix `m` has elements `m(i,j)`. Another difference is that while the literature names are usually restricted to one character (usually English or Greek), MATLAB matrix names can have several characters. As a result, MATLAB's notation is a bit less compact than the literature notation, in that multiplication cannot be expressed as `ab` (which may be the name of a single MATLAB variable), but instead requires an intervening operator `a*b`. These conventions enable one to code in linear fashion, using text editors that do not handle subscripts, and it allows variable names and general layout to be similar to other languages like FORTRAN and C++.

The matrix transpose a' denotes that the rows of a become columns and the columns become rows, e.g., `t = a'` means `t(i,j) = a(j,i)` for all `i` and `j`.

```
>> a=[ 1,2,3;  4,5,6;  7,8,9 ];
>> t=a'
   t = 1 4 7
       2 5 8
       3 6 9
```

1.4 Matrix multiplication

The major difference between scalar and matrix algebras lies in the definition of matrix multiplication, which can be viewed in several different ways. The product `c = a*b` can be defined element by element in `c`. Let `a` be an `m` by `q` matrix, and let `b` be a `q` by `n` matrix. Then `c` is an `m` by `n` matrix with elements:

```
c(i,j) = a(i,1)*b(1,j) + a(i,2)*b(2,j) + ... + a(i,q)*b(q,j)
```

So each `c(i,j)` is a sum of products using elements from the ith row of `a` and the jth column of `b`. Hence, `a` must have the same number of columns as `b` has rows, or some of those products will be undefined. As an example:

```
>>a = [ 1,2;
        3,4;
        5,6 ];
>>b = [ -1, -2, -3, -4;
         4,  5,  6,  7 ];
>>c = a*b
  c =   7  8  9 10
       13 14 15 16
       19 20 21 22
```

Every element of `c` is the sum of two products, e.g.:

```
c(2,3)  =  3*(-3) + 4*6  =  15
```

Note that the product `b*a` is not defined because `b` has more columns than `a` has rows.

Matrix multiplication also defines two kinds of vector multiplication as special cases. The first is called the inner product, between a row vector on the left and a column vector on the right. The two vectors must have the same number of elements, and the result is a scalar (i.e. a 1 by 1 matrix). For example, if `c` is a column vector, then the product `c'*c` is the sum of squares of the elements in `c`. Denoting the `i`th row of `a` as `a(i,:)` and the `j`th column of `a` as `a(:,j)`, the matrix product can now be expressed as a set of inner products:

```
c(i,j)  =  a(i,:)*b(:,j)
```

The second kind of vector product is called the outer product, with a column vector on the left and a row vector on the right, with no restrictions on their lengths. The result is a matrix, usually with several rows and columns. If `a` has `m` rows and `b` has `n` columns, then the product `a*b` is an `m-by-n` matrix, and each element is formed by a product of two scalars. A matrix product can now be expressed as a sum of outer products of the various rows and columns:

```
c = a(:,1)*b(1,:) + a(:,2)*b(2,:) + more terms if needed.
```

There are two other ways to look at matrix multiplication. Consider multiplying a matrix `B` by a row vector `a` on the left. The product is a single row which is a linear combination of the rows in B:

```
p = aB = a(1)*B(1,:) + a(2)*B(2,:) + ... + a(n)*B(n,:)
```

Note that a matrix is (among other concepts) a stack of rows like `a`. Call the entire stack `A`. Now the product `P=AB` is simply a stack of rows like `p` above, each row of `P` corresponding to a row of `A`. Like the row `p`, each row of `P` is a linear combination of the rows of `B`:

```
P(i,:) = A(i,:)*B = A(i,1)*B(1,:) + A(i,2)*B(2,:) + ...
```

By entirely analogous reasoning, we note that the columns of `P` are linear combinations of the columns of A, i.e.:

```
P(:,j) = A(:,1)*B(1,j) + A(:,2)*B(2,J) + ...
```

All this means that the rows of `P` cannot be outside of the space defined by the rows of `B`, and the columns of `P` cannot be outside of the space defined by the columns of `A`. For example, if all rows of `B` are identical, all rows of `P` will have that same shape.

In summary, a matrix product `AB` may be viewed as:

1. A sum of outer products `A(:,i)*B(i,:)`.
2. A matrix of inner products `p(i,j)=A(i,:)*B(:,j)`,
3. A set of columns, each a linear combination of the columns of `A`.
4. A set of rows, each a linear combination of the rows of `B`.

It may seem like quibbling to express the same operation in all these different ways, but it is very useful quibbling. Such reexpression is of assistance in proving properties of operations, in optimizing efficiency (computer architectures will influence the style of calculation), and in the control of roundoff error. Even physical insights will be enhanced by one form more than another. Consider e.g. a reacting chemical mixture observed by absorbance spectrum at many wavelengths and reaction times. The wavelength vector `w` has `m` elements `w(i)`, the time vector `t` has `n` elements `t(j)`, and the observed absorbance matrix `a` is therefore `m-by-n`, with elements `a(i,j)` corresponding to `w(i)` and `t(j)`. Each chemical species has an absorbance spectrum stored in some column of the matrix `b`, and its concentration is a function of time stored in the corresponding row of `c`. By expressing the relation `a=b*c` as a series of terms `b(:,j)*c(j,:)`, each `j`th term represents the contribution of the `j`th chemical species to the observed spectra in `a`. Even if one does not know `b` or `c`, recognizing the relation `a=bc` can enable one to create mathematical models that will lead to the generation of `b` and `c` by numerical procedures.

1.5 Scalar *versus* matrix algebras

Returning now to the relation between scalar and matrix algebras, recall that the scalar operations addition and multiplication have laws by which they may be manipulated:

Law	Addition	Multiplication
Closure	$a+b=c$	$a*b=c$
Associative	$(a+b)+c=a+(b+c)$	$(a*b)*c=a*(b*c)$
Commutative	$a+b=b+a$	$a*b=b*a$
Identity	$a+0=a$	$a*I=a$ and $I*a=a$
Inverse	$a+(-a)=0$	$a*(1/a)=I$
Distributive	$a*(b+c)=a*b+a*c$ and	$(a+b)*c=a*c+b*c$

The laws are stated in mnemonic rather than rigorous form, so that some discussion is required to devine what the laws mean. Most of these laws are obeyed by matrices as well as scalars, provided the matrices are conformal (i.e. have the right dimensions) for the operations involved. Closure means that an operation on two scalars (matrices) produces another scalar (matrix). The associative law means that in a long string of e.g. additions, the operations can be done in any order. However, this law does not give one permission to disturb the order of the operands. That is, rearranging the product `a*b*c*d` to `a*d*c*b` is not granted by association, but rather by commutation, and it is important to note that *matrix multiplication does not have a general commutative law.* For this reason among others, you will need to be cautious about rearranging matrix expressions until you grow accustomed to restrictions on the various operations.

One major issue in rearranging multiplications is efficiency. The number of flops (floating point operations) in the product of an `m-by-q` matrix and a `q-by-n` matrix by the above methods is `mn(2q-1)`. Since the matrices in a product can be of vastly different sizes, it pays to optimize the order (in the associative sense) in which the work is done. For example, let `A, B`, and `C` represent 50 by 50 matrices, and let `v` be a column 50-vector. Then the calculation `((A*B)*C)*v` uses `499,950` flops, whereas `A*(B*(C*v))` uses only `14,850` flops.

1.6 Matrix inversion

Aside from the non-commutativity of matrix multiplication, the other major difference between scalar algebra and matrix algebra lies in the nature of an inverse. (In matrix parlance, the term "inverse" means multiplicative inverse.) Matrices corresponding to the scalar zero are matrices of all zeros (of any size), i.e., zero matrices. When we say `b=0`, we mean `b` contains all zeros, and `b` is the only matrix for which `a+b=a` holds. Furthermore, the negative of `a`, denoted `-a`, contains the scalar negatives `-a(i,j)` as elements, so zeros and negatives of matrices are easy to form. But (multiplicative) inverses are usually not easy to form, nor is it usually easy to tell if a matrix has an inverse.

In order to define "inverse", we must first define the (multiplicative) identity matrix I, which is simply a square matrix with ones on the main diagonal and zeros everywhere else. For example, a 3 by 3 indentity matrix is:

```
I = [ 1, 0, 0;
      0, 1, 0;
      0, 0, 1 ];
```

The standard symbol for an identity of any size is `I`. You can verify that for square matrices `a` and `I` (of the same size), `a*I=I*a=a`. If `a` has an inverse `b`, then `a*b=b*a=I`, and `a` is called nonsingular, otherwise `a` is called singular. Singularity is a very important concept that bears on the solvability and uniqueness of solutions of simultaneous linear equations. To illustrate the difficulty of recognizing a singular matrix when you see one, the following 3 by 3 matrices are all singular:

```
0 0 0    1 2 3    1 1 1    1 2 3    1 1 1
0 0 0    1 5 3    1 1 1    2 4 6    1 2 3
0 0 0    0 0 0    1 1 1    3 6 9    2 5 8
```

When solving one scalar linear equation `a*x=y` for the single unknown `x`, the solution is unique whenever the scalar `a` is nonzero. By contrast, the matrix equation `a*x=y` can represent `n` scalar equations in `n` unknowns. In that case, `x` and `y` are `n`-vectors, `a` is an `n-by-n` matrix, and the equations can be solved uniquely if and only if `a` is nonsingular. Deciding which matrices are "practically" singular can involve several numerical and conceptual issues like tolerances, scaling, and noise.

The goal of this chapter has been to give a rudimentary impression of the differences between scalar and matrix manipulations. I recommend that you read the early sections of a standard text. One of the best is: Linear Algebra and Its Applications, by Gilbert Strang, third edition, 1988, Harcourt Brace Jovanovich, San Diego. A matrix text with emphasis on MATLAB is: Matrices and MATLAB, by Marvin Marcus, 1993, Prentice Hall, Englewood Cliffs NJ.

Chapter 2

Why MATLAB?

2.1 A little background

Originally, MATLAB was conceived as an academic tool, giving professors and students access to first-class numerical software, especially in the area of linear algebra. With the formation of The MathWorks, Inc. in the early 1980's, and the consequent sale of MATLAB as commercial software, pressures from clients began to build for an expansion of capabilities. Today, MATLAB's popularity has grown far beyond the academic community, and the language has grown far beyond its original role as a teaching tool. While there are still hangovers from the days when statements were usually issued and executed one at a time (e.g. results are still printed if you omit the semicolon), MATLAB is now a fully-formed programming language, capable of many kinds of information processing, including the advanced design of user interfaces. Unfortunately, several numerical capabilities, like optimization, signal processing, and partial differential equations, are sold as separate tool boxes, like the omnipresent batteries for children's toys, only more expensive. (A considerable part of the signal processing toolbox was once part of MATLAB proper, but has since been split off.) One can only hope that The MathWorks will eventually see fit to unite the numerical-analysis toolboxes under the central MATLAB umbrella.

2.2 Language traits

MATLAB has several traits that make it a pleasure to use:

- MATLAB has an algebra-like syntax, so that statements tend to look like what they do. This means that a user can return to a project after weeks of distraction and quickly recall what his programs are about. (Of course, a generous supply of user's comments are always welcome in this regard. My own programs contain more comments than code.) MATLAB code is both intelligible and compact, a welcome compromise between the verbose, loop-filled style of FORTRAN and the cryptic density of APL.

- MATLAB is readily vectorizable. Vector hardware enhances the performance of many current computers. Since MATLAB tends to say things a vector or matrix at a time,

it is natural for a MATLAB implementation to take advantage of any vector features the machine may have, without changing its appearance to the user.

- MATLAB is interactive. This is no novelty these days, but it is important. A nontrivial program can be written, debugged, and run all within minutes, including graphic output. Despite the use of interpreted code, as opposed to compiled languages like C++ or FORTRAN, execution is surprisingly efficient when one takes full advantage of vector-based notation, even on a non-vector machine.

- When things go wrong, MATLAB usually provides informative error messages, and enables you to do as well with your own code. The debugging facilities of the language are plentiful and easy to apply.

- The graphic facilities are extensive, catering to all levels of expertise, from the casual user, who just wants to see the curve, to the most artistic of animation mavens.

2.3 The function call: a case in point

The clarity of the MATLAB language is exemplified by the function call. Consider a FORTRAN-style subroutine call circa 1970:

```
CALL MATMUL(a,b,c,d,e,f,g)
```

Several problems confront a user looking at this statement. Which arguments are input? Which arguments are output? Which arguments are important pieces of data like matrices, and which are mere bookkeeping items that the computer should have figured out for itself, like the sizes of matrices, the sizes of storage areas, and temporary storage for holding arrays of unpredictable size? Which of the input arguments will be changed on output? Has the caller provided sufficiently large arrays to hold the intermediate and output results?

Contrast the above FORTRAN call with the MATLAB call:

```
[u,s,v]=svd(a,kopt);
```

The outputs are `u`, `s`, and `v`. The inputs are `a` and `kopt`. The inputs are never altered (unless they are also in the output list). Instead, when functions alter input data, copies are made which may be altered with no effect on the originals. Temporary variables required by the function are allocated dynamically within the function. The caller is not concerned with them. The sizes of input matrices are known to the system, so the caller is never obliged to provide them explicitly. The sizes of output matrices are governed by the function, and need not be pre-allocated by the caller. In short, a MATLAB function call contains only the things that count, it is crystal clear which arguments are input and which are output, and there are no side effects unless you ask for them, e.g., plots, prompts, or altered global variables. (In a pinch, there are ways to circumvent this neatness. See the function `evalin` described in the section on workspaces in the chapter on M-files.)

Chapter 3

The Look and Feel of MATLAB

Matlab is a high-level language for computing with numerical arrays and other data types. But before we get into that aspect of it, there are several features of the language that are not computing *per se*, but which do make your life easier.

3.1 Start, stop, and emergency stop

First, go to your working directory, or be sure that your file-access path will let you get to the files you need. This is a system issue, and you should consult with your local computer staff, if any, for approved procedures. To enter MATLAB, say `matlab`, or whatever convention has been set up on your computer. To get out of MATLAB, say `exit`. Once in MATLAB, if you create a monster process that threatens to run forever or produce gobs of unwanted output, you can interrupt that process with control-c, which will return control of the session to you in most cases.

3.2 Creating a diary

A diary is an ASCII file listing all the commands that you have issued during a session, along with MATLAB's printed responses to them. The command `diary on` starts the diary. You can say `diary off` at any time in your session, do things you don't want in the diary, then say `diary on` again when you want to resume. I prefer to leave the diary on during the session, then if necessary, edit the diary file after the session. That way, I don't forget to turn the diary back on again. Also, I'm never sure what might prove important during the heat of computation, the thrill of victory, or the agony of defeat.

Forgetting to start a diary in an otherwise productive session can be most frustrating. To insure that your diary will be on at the outset, put a `diary on` statement in your startup.m file, which MATLAB calls upon entry, so you can't forget it. In MATLAB 5, the command `exit` will save the diary if one exists.

3.3 Functions and script files

In general, commands to MATLAB may be issued directly by you, or by M-files (files named with the suffix `.m`) that you invoke. There are two styles of M-file. The simplest is the script file. When you invoke a script by typing its name without the `.m` extension, the statements it issues are indistinguishable from the statements you issue at the terminal. Variables defined by you or that script file are available to you and the script file. This convention requires some care in choosing variable names, to avoid inadvertent altering of essential values. By contrast, function files communicate (mostly) through input and output arguments. Variables within a function are private to that function, and to the script files that it invokes. (Scripts are fickle in that they use the variables of any process that invokes them.) To illustrate, let's look at a session that uses a script file and a function.

```
% ----- This comment begins the function file hyperb.m -----
function Y=hyperb(X)
X=2*X;   % This X has no bearing on X in the master session.
Y=1 ./ (1+X);   % Same idea for Y.
% ----- This comment ends the function file hyperb.m ------

% ----- This comment begins the script file double.m, -------
% ----- which is invoked only by the master session.
X=2*X;   % Master session's X is doubled.
% ----- This comment ends the script file double.m ----------

% ----- This comment begins the master session -----
X = [1,1,2,3,5,8];   Y = X+5;
Z = hyperb(Y);   % This statement invokes the function file.
                 % Master session X & Y remain unchanged.
                 % Only Z is affected.
double;          % This statement invokes the script file.
                 % Master session X is now [2,2,4,6,10,16];
```

3.4 Statement delimiters

MATLAB accepts executable statements from the keyboard, or from a file, one line at a time. For example, if the statement:

```
y = ( x^2 + 5*x + 6 )  ...
  / ( x^2 + 6*x + 7 );
```

is entered with the "..." omitted, the result of line 1 is stored in y and printed (because MATLAB prints the result of any statement that doesn't end in ";") and line 2 is regarded as a separate but malformed statement, producing an error. To continue a statement over two or more lines, every non-final line must end in an ellipsis "...". For exceptions, see the chapter on tricky stuff. The ellipsis is important because some statements will not fit on a line, and because it also gives the user the option to make certain expressions look clearer. All text to the right of an ellipsis is ignored.

Ending a statement with ";" has two nice effects. First, it suppresses those mostly unwanted printouts of results. Second, it enables the entry of more than one statement on a line. Multi-statement lines should be used judiciously. Indiscriminate jumbling of statements can destroy a program's readability, but proper use enables one to group statements a natural way, and avoid sprawl.

3.5 Blanks and comments

Other features that improve readability are the blank and the comment. Blanks may be inserted anywhere except in the middle of a name (e.g. `delta`, but not `del ta`) or operator (e.g. `a .* b`, but not `a. *b`), or number (e.g. `1e6`, not `1 e6`). For exceptions, see the chapter on tricky stuff.

The character `%` anywhere on a line tells MATLAB to ignore the rest of the line, including other `%`'s. Thus the above example can be commented as follows:

```
% -- Compute the %!@#$&* rational function --
y = ( x^2 + 5*x + 6 ) ...   % Numerator.
  / ( x^2 + 6*x + 7 ) ;     % Denominator.
```

Notice that whole or partial lines can be comments, but an executable statement can never appear to the right of a comment.

3.6 Temporary exits

Sometimes, in the middle of a MATLAB session, you may want to edit or examine a file, or look at a directory, or do some non-MATLAB computing to set up the next stage of the session. Yet, you do not want to lose what has been done so far. One way to do this is to save everything you have defined in some file, exit MATLAB, do your chores, reenter MATLAB, retrieve the file, and optionally, delete the file when you finish. The `save` and `load` commands will allow you to do this with very little typing:

```
save tempfile;   exit;
Do your non-MATLAB chores here.
matlab
load tempfile;
!del tempfile
```

You are now ready to resume the session. For most versions of MATLAB, there is a more efficient way to do non-MATLAB chores within a MATLAB session. Any command that begins with `!` is not executed by MATLAB, but is instead referred to the operating system (see e.g. the last line of the above example):

```
!edit myfunc.m
```

The above command will be executed by the system, not by MATLAB, and the file myfunc.m can then be edited. Terminating the edit causes a return to MATLAB with the session intact. Commands beginning with `!` can invoke any handy editor, or do anything understood by the system.

3.7 Correcting commands

As you are typing a command, mistakes can be corrected by backspacing etc. in the usual manner. Mistakes in a previous command can be corrected easily by hitting the "up-arrow" key, which recalls the previously-entered line for editing and/or reentry. Likewise, by hitting "up-arrow" repeatedly, any previously-entered line can be recalled. As a shortcut, type the first few characters of the line you want to retrieve, then "up-arrow" will access only lines that begin with those characters.

3.8 Partial summary

To summarize the features so far, you can:

- start a MATLAB session,
- exit from a MATLAB session,
- stop a runaway process,
- create a diary,
- issue statements directly, or from M-files,
- exit temporarily and reenter without losing anything,
- suppress printout using semicolon,
- put more than one statement on a line,
- spread one statement over several lines,
- put comments to the right of "%" symbols,
- insert blanks in most reasonable places,
- recall previous lines for editing and reuse.

3.9 Manipulating matrices

The five basic infix operations (+,-,*,/,^) and many functions (e.g., log and sin) are available for doing scalar arithmetic, which looks much the same as in FORTRAN or C.

```
a=exp(b)*sin(c)^log(d);   % a,b,c,d are scalars.
```

Elements of matrices can be treated as scalars by subscripting.

```
X(2,2)=exp(X(1,1))*sin(X(1,2))^log(X(2,1));
```

But the real value of MATLAB is its ability to treat a whole matrix as a notational unit, thus doing away with much of the looping found in lower-level languages. A matrix can be created explicitly, using square brackets. Elements of a row are delimited by commas or spaces, and rows are delimited by semicolons or end-of-lines.

```
X=[ 1,2,3; 4 5 6 ];   % X has 2 rows, 3 columns.
Y=[ 1 2 3             % Y is the
    4,5,6 ];          %    same as X.
```

Square brackets can also catenate matrices to form larger matrices.

```
Y=[ X,X; X,-X ];   % Using X from the above example,
                   % Y becomes [ 1 2 3   1  2  3
                   %              4 5 6   4  5  6
                   %              1 2 3  -1 -2 -3
                   %              4 5 6  -4 -5 -6 ]
```

Matrices of arbitrary size can be generated.

```
u=0:.2:6;   % u is a row vector of values 0 to 6 in steps of 0.2.
Y=zeros(size(X));   % Y is a matrix of zeros, same size as X.
Z=randn(m,n);   % Z is an m-by-n matrix of random numbers.
```

Matrices can also be read from files. The following is a code for reading an ASCII file of numbers with arbitrary format into a vector v.

```
fid=fopen('matfile');   % Open the file matfile.
v=fscanf(fid,'%g');   % Read all nos. from matfile into vector v.
status=fclose(fid);   % Close matfile (which remains unchanged).
```

Parts of matrices can be extracted or deleted. Here is code for extracting, then deleting rows 2, 3, and 5 from an existing 6-by-6 matrix X:

```
v=[2,3,5];    % v is a vector of indices.
Y=X(v,:);     % Y gets X rows 2, 3, & 5.   X is unchanged.
X(v,:)=[ ];   % X is now 3-by-6, with rows 2, 3, & 5 deleted.
```

All manner of operations can be performed with pre-existing matrices.

```
% ----- Let a,b,c be scalars.
% ----- Let r,s.t.u,v be vectors of appropriate size.
% ----- Let W,X,Y,Z be matrices of appropriate size.
W=sin(X);   % Every element W(i,j)=sin(X(i,j)).
Z=X*Y;      % Matrix product.
Z=X.*Y      % Array product: every Z(i,j)=X(i,j)*Y(i,j).
Y=X^2       % Y gets X*X, matrix 2nd power.
Y=X.^2      % Every Y(i,j)=X(i,j)^2.
% ----- In the following 3 statements, Y gets X inverse
% ----- when X is square and nonsingular.
Y=eye(size(X))/X;   Y=inv(X);   Y=X^(-1);
% ----- Functions can accept and produce any combination
% ----- of scalars, vectors, and matrices.
[X,Y,Z]=svd(W);   % W is factored into special matrices X.Y,Z.
% ----- Linear least-squares is easy in MATLAB.
u=X\v;   % u gets the least-squares solution to Xu=v.
s=X*u;   % s is the approximating function to v.
r=v-s;   % r is the residual vector for the above solution.
b=sum(r.*r);   % b is the sum of squares for the above solution.
% ----- If t is the independent variable vector for the above
% ----- problem, the following plots may be of use.
plot(t,r); % Plot the residuals r vs. t.
semilogy(t,v,t,s); % Compare data v & model s on a semilog plot.
```

3.10 Summary of array operations

Numerical arrays are generalizations of matrices, in that they can have an arbitrary number of indices.

1. Arrays can operated upon in one-element-at-a-time style, i.e., incremented, decremented, multiplied, divided or exponentiated by a scalar (scalar-by-matrix operations), or by corresponding elements of another matrix (array operations).

2. In similar style, scalar functions like `sin` and `log` can be applied to arrays in one step, producing an element-for-element result.

3. One must be careful to distinguish between the above array-style operations and matrix operations like matrix product, matrix inverse, and matrix exponentiation, which are not element-for-element.

4. Arrays can be created using file-reading statements, array-initializing functions like `zeros`, colon notation for equally-spaced vectors, and square-bracket notation for arbitrary matrices.

5. Arrays can be built from smaller arrays using catenation, i.e. square brackets and the `cat` function.

6. Arrays can be rearranged using the transpose operator ('), various functions such as `reshape` and `flipud`, and indexing such as `v([3,2,1])=v(1:3)`, which reverses the order of the first 3 elements of `v`.

7. An array can be augmented by storing numbers in non-existing elements, as in `v(7)=0` where `v` had only five elements initially.

8. Cross-sections of arrays can be removed by storing null information in them, e.g. so that `X(:,3)=[]` removes the third column of `X` (the fourth column becomes the third column and so on).

3.11 Program flow control

3.11.1 The for-loop

Matrix conventions avoid many loops, but not all loops. A generic MATLAB for-loop looks like this:

```
for c = X;
   % Statements within the loop go here.
end;
```

If `X` is null, the for-loop is bypassed. If `X` is a row n-vector, the scalar `c` will be assigned each element of `X` in turn, and the for-loop will be executed n times. If `X` is an m-by-n matrix with $m > 1$, the vector `c` will be assigned the columns of `X` in turn, and again, the for-loop will be executed n times. (Note that if `X` is a column, the loop is executed only once, with `c=X`.) A for-loop can be terminated before n iterations are completed, by using a `break` statement or a `return` statement within the loop.

3.11.2 The while-loop

The while-loop is iterated as long as a given logical scalar expression (see "logical variables" below) is true, i.e.:

```
while L1, (Repeat this part as long as L1 is true); end;
```

The while-loop has an inherent risk. Unlike the for-loop, there is no built-in guarantee that the loop will terminate, i.e. infinite while-loops are possible. It is up to the programmer to be sure that there is a stopping condition.

3.11.3 The if-then-elseif-else construct

The `if` statement is similar to that in FORTRAN, e.g. let `L1`, `L2`, `L3`, etc. be logical scalar expressions:

```
if L1, y=f1(x); % This section is used if L1 is true.
  elseif L2, y=f2(x); % Used if L1 is false and L2 is true.
  elseif L3, y=f3(x); % if L1 and L2 are false and L3 is true.
     ... more elseif's if needed ...
  else y=f3(x); % Used if L1, L2, L3, etc. are false.
end;
```

Only the `if` block is required. The `elseif` and `else` blocks are optional.

3.11.4 The switch-case construct

The `switch` statement allows a multi-branched decision in concise form:

```
switch e0;   % e0 is some expression.
   case {a1,a2};      y=fa(x);   % used if e0=a1 or a2.
   case {b1,b2,b3};   y=fb(x);   % used if e0=b1, b2, or b3.
   case c1            y=fc(x);   % used if e0=c1.
   % More cases if needed; otherwise (below) is optional.
   otherwise;         y=f(x);    % used if no match for e0.
end;
```

At most one case will be executed, after which control passes to the statement after `end`. If there is no match for e0, and the `otherwise` clause has been omitted, then the `switch` construct is simply bypassed. The `switch` clause is the only required clause. The `case` and `otherwise` clauses are optional. However, it is pointless to use the `switch` construct without `case` clauses, and it is often risky to omit the `otherwise` clause.

3.11.5 The try-catch construct

Ordinarily, when a detectable error occurs, execution is stopped, an error message is printed, and control is returned to the user. However, there are times when errors are anticipated, and the user wishes to code an automatic reaction to them. The construct for automatic error trapping is:

```
try;
   % Error-prone block of code goes here.
   % If an error occurs,...
catch;
   % ... control is transferred here.  This is
   % a block of code that reacts to the error.
end;
```

A string called `lasterr` receives the error message, execution of the `try` block is discontinued at the point where the error occurred, and control is transferred to the `catch` block, which can use the string `lasterr` to figure out what to do. One can nest try-catches much as one can nest if-elseif-elses. The error message is not printed unless there is an explicit statement to do so. Currently, the `catch` block is optional, which is dangerous. If you code a `try` block without a `catch` block, an error can occur with neither an error message nor a corrective action.

3.12 Kinds of data

Numerical matrices are the data that MATLAB was originally designed for. In its current incarnation, MATLAB has several ways of classifying, organizing, and manipulating data. There are no declarations of type in MATLAB such as the DECLARE or DIMENSION statements in FORTRAN. A variable takes on the type of expression that is assigned to it, and during the variable's life, it can change in value, size, and other ways. Variables, or parts of them, can be thought of as numbers, 0-1 logic, bits, bytes, characters, arrays, cells, structures, and objects.

3.12.1 Arrays

All built-in data types in MATLAB are arrays. An array is a collection of data arranged in rectangular fashion, with rows, columns, pages (matrices), etc. Several arrays can be catenated into a larger array using square brackets or the `cat` function. Parts of an array can be accessed, deleted, or altered using indices in parentheses. Methods for doing these things with numerical arrays are discussed in the next few chapters. Other kinds of arrays are used in similar fashion.

3.12.2 Numerical arrays

Numerical arrays contain either double-precision floating-point numbers, or 8-bit integers. Arrays may be scalar (1-by-1), vector (n-by-1 or 1-by-n), matrix (n1-by-n2), or of higher dimension (n1-by-n2-by-...). For example, the statement `A(1,2,3,4)=5;` stores the value `5` in the first row, second column, third page, fourth book, if you will, of the array `A`. Indexing rules for matrices and arrays are discussed in the next three chapters of these notes.

3.12.3 Sparse arrays

Sparse arrays are specialized numerical arrays. For applications where large matrices contain mostly zeros, the sparse data type may be of use. In a sparse matrix (dimensions beyond the second are not allowed for sparse data), only the nonzero elements are stored, along with their indices. This may save an immense amount of storage, and may also permit faster processing in fortunate cases. Like the full (non-sparse) storage format, sparse matrices are stored in double precision, and may also be logical or complex (see below).

While sparse format can save workspace, MATLAB must store indices along with nonzero entries, so each nonzero takes more space (16 bytes per real, 24 bytes per complex). For example, if a real matrix is half zeros, you save no space by using sparse format. Also, sparse indexing and many sparse numerical methods can have greater computational overhead (e.g. geater access time per element) than their full counterparts. Often, there is a crossover in the efficiency curves of full and sparse methods, depending on array size and fraction of zeros. Sometimes, the need for sparse format is overwhelming, but if not, some experimentation may be in order to find the best format for your case. See the chapter on time and work.

3.12.4 Logical arrays

Logical data types are a subset of numerical data types, and indeed they can be used as numbers. But they also have a special interpretation and an enriched set of operations. Logical arrays are used with great dexterity and compactness in MATLAB, not only for controlling program flow in `if` statements and while-loops, but also for indexing and loop avoidance. The relational operators `( <, <=, ==, ~=, >=, > )` work in array fashion. In an expression like `a>X` where `a` is scalar and `X` is a matrix, the result is a logical (0-1) matrix the same size as `X`, which contains a 1 wherever `a>X(i,j)` holds, and 0 otherwise. Alternately, `Y>X`, where `X` and `Y` are matrices of the same size, produces a logical matrix of the same size containing a 1 wherever `Y(i,j)>X(i,j)`, and 0 otherwise. These logical matrices can in turn be used as indices to access or alter pertinent parts of a matrix, e.g.: `v=u(u>5)` copies the elements of vector `u` which are greater than 5 into the vector `v`. Logical indexing will be discussed in detail in the next two chapters.

Logical variables can also be created from numerical variables by a simple conversion function: `L=logical(A)`. The logical variable `L` has the same size as `A`, *and the same values.* However, when used in a logical context, the entries of `L` are interpreted as "false" when the value is 0, and "true" when the value is any other number.

3.12.5 Complex arrays

Complex arrays are a subclass of numerical arrays. A complex number is stored as a pair of double precision numbers, namely, the real and imaginary part, occupying 16 bytes per number (24 bytes per number in sparse format). Operations on real and complex data look alike, which can be disconcerting at times. For example, `sqrt(x)` will work even for `x<0`, producing an imaginary number when you might have preferred an error message. If an array is real (no imaginary parts),and you store a complex value in one of its elements, the entire array is converted to complex format. MATLAB makes some effort to convert back to real format when all imaginary parts become zero.

MATLAB functions of particular interest in complex arithmetic (see MATLAB help) are:

```
isreal  real  imag  conj  angle  abs
```

There are many other MATLAB functions that are valid for both real and complex data, such as elementary functions like `sin` and `exp`, special math functions like the Bessel functions, and matrix operations like `inv` and `eig`.

3.12.6 Bits

There is not really a separate data type for bits, but there are several MATLAB-provided functions that treat numerical data in bitwise fashion, enabling you to store, retrieve, and alter bits as though there were such a data type. These functions are:

```
bitand bitcmp bitor bitmax bitxor bitset bitget bitshift
```

These functions operate on the integer (mantissa) portions of double-precision floating-point numbers, allowing 52 bits per number rather than the full 64. Specifics may be had from the help file by saying, e.g., `help bitor`.

3.12.7 Bytes

In MATLAB, `uint8` (byte or 8-bit) data is regarded as a subclass of numerical data, although arithmetic operations are not defined for bytes. Bytes, interpreted as 8-bit integers, are used to keep image storage under control. Images are notorious for their sheer bulk in computer memory and disk storage. The function `uint8` converts the usual 64-bit numbers to 8-bit integers, but it is up to the user to insure that those numbers are in the range 0-255. MATLAB does not currently specify what will happen if the numbers are out of range. The results may be platform-dependent.

Like double precision data, `uint8` arrays may be real, logical, or complex. The indexing of 8-bit arrays is like that of numerical arrays. However, 8-bit data does not share all the priviledges of full precision. 8-bit data has no sparse format. Because arithmetic for 8-bit data is not provided, one must convert to 64-bit format using the `double` function, do the arithmetic, then convert back using `uint8`.

MATLAB provides several tools for handling 8-bit data. These may be found under the help-file entries:

```
uint8 double image imagesc infinfo imread imwrite
```

3.12.8 Characters

A character is stored in 16-bit format containing the ASCII code for that character, e.g. `'d'` will be stored as 100, although you will seldom care about its numerical value. Character strings such as `'Hi there'`, are stored as row-vectors. You can build higher dimensional character arrays in the same manner as with numbers, the major restriction being that all strings (rows) in such an array must be of equal length, so that shorter lines must be padded with blanks. This restriction can be avoided by storing character strings in cells (see below) rather than arrays. MATLAB provides several tools for manipulating character data, and for processing numbers in character form. These are found under the help-file entries:

```
base2dec bin2dec char dec2base dec2bin
mat2str strcat strmatch strncmp strvcat
```

3.12.9 Structure and cell arrays

Cells, structures, and their arrays may hold data of nonuniform type and size, including other cells and structures. The variety of possible data organization using cells and structures is truly bewildering. The MATLAB manual "Using MATLAB" (version 5) has a chapter detailing their use. A later chapter "Cells and Structures" in these notes will also be useful. MATLAB provides functions for handling them, as explained in the help-file entries:

```
   cell cell2struct celldisp cellplot deal num2cell
fieldnames getfield rmfield setfield struct struct2cell
            iscell isfield isstruct
```

3.12.10 Classes and objects

Objects are structures with special rules of manipulation that depend on the class to which the objects belong. There are five built-in classes in MATLAB: `double`, `sparse`, `char`, `struct`, and `cell`. Rules for your own classes are user-defined by files stored in specially-named directories. For example, the directory `@myobj` contains rules for objects of class `myobj`. Within that directory is a special file named `myobj.m` (same as the directory but without the `@` sign) known as a constructor. The constructor's job is to define new objects of the class `myobj`. Other files in the `@myobj` directory define the rules of manipulating objects of class `myobj`, including conversion to objects of other classes. Objects of a "child" class may inherit attributes from several "parent" classes, so that methods from a parent class can operate on appropriate parts of the "child" object. The rules can redefine operators like `+` and `*` so that something nonstandard happens to `A+B` when either `A` or `B` is an object. The MATLAB manual "Using MATLAB" (version 5) has a chapter on classes and objects. Also, MATLAB provides the following entries in the help-file:

```
  class  struct  methods  isa
isobject  inferiorto  superiorto
```

Also see the help entry `datatypes` under the heading "Overloadable operators" for those built-in operators that may be overwritten (i.e. redefined) by object-oriented methods.

Chapter 4

Indexing

This chapter discusses indexing, which is the most characteristic operation of the MATLAB language. In the early sections, we will consider cases where only one or two indices are involved. By fairly direct extension, the rules for two indices apply to three or more indices, a topic considered in the later sections of this chapter. Also, the early sections refer to numerical matrices, but the same ideas are used to index general arrays of characters, cells, structures, and user-defined objects.

4.1 Sources, intermediates, and targets

A source is any pre-existing array from which we are extracting information, whereas a target is any matrix (pre-existing or not) into which we are storing information. In the following example, there is a single target `T`, and six sources:

```
T(K1,K2) = U*S(K3,K4);   % Sources: K1,K2,K3,K4,U,S.
```

However, when MATLAB executes this statement, it also produces a series of intermediate results, e.g:

- `R1,R2,R3,R4` are versions of `K1,K2,K3,K4` suitable for indexing.
- `R5` is the result of `S(K3,K4)`.
- `R6`, the final intermediate, is the product `U*R5`.

The final intermediate `R` may then have to be converted, e.g. reshaped, to fit into the appropriate part of `T`. In addition, `T` itself may be reshaped by its indices and by the shape of `R`. Sources themselves are never altered unless they are also targets.

Only sources and targets can be indexed. Implied intermediates can not be indexed, making the following example illegal:

```
T(J) = (U*S)(K);
```

Here we have a failed attempt to index the matrix product `U*S`. Even if `K` is a valid index of the matrix `U*S`, the process must be done in two statements:

```
X=U*S;   T(J)=X(K);
```

4.2 Definitions: puts, gets, and vec

The symbol `->` will mean "puts" or "produces" in the sense that an indexed source `S` puts an intermediate result in `R`. The symbol `<-` will mean "gets or "becomes" in the sense that an indexed target `T` gets the final intermediate `R`.

MATLAB's indexing conventions often require arbitrary arrays to be regarded as vectors. The `vec` operator changes an array to a column vector, by stacking successive columns below the first, which puts the elements in column-major order:

```
vec([1,2,3; 4,5,6]) -> [1; 4; 2; 5; 3; 6]
```

We will also use "vec" as a verb (forms: vec, vecking, vecked). The `vec` operator is not an explicit MATLAB-supplied function, but it is very easily coded as `M(:)`. For instructional convenience, we can code this operation as a MATLAB function:

```
function V = vec(M);   V = M(:);
```

For indexing, we usually have no need to vec anything explicitly. MATLAB does it as required, e.g. whenever a matrix or array source `S` is singly-indexed, as in `S(J)` rather than, say, `S(J,K)`, `vec(S)` is used implicitly:

```
S = [1,2; 3,4];   S(3) -> 2
```

Vecking a matrix involves almost no computation or number-shuffling. MATLAB already stores the entries in column-major order. For example, to change a 30X20 matrix to a column 600-vector requires only that the dimensions be changed from 30X20 to 600X1. The entries of the matrix remain stored in the same order.

4.3 Source indexing *versus* target indexing

4.3.1 Sources

The purpose of indexing a source `S`, as in `S(J,K)->R`, is to form an intermediate R from the parts (elements, rows, etc.) of S, with possible permutations and repeats. `S` itself is never altered. Initially, `R` does not exist. It is born when `S` is indexed, and it dies upon further use. This short-lived value of `R` depends only on `S` and its indices. Unlike indexed targets, indexed sources (and sources in general) must pre-exist. If `S` does not exist, an assignment like `T=S(:)` will not produce some default `S` to store in `T`; it will produce an error. Likewise, index entries must refer to parts of `S` that exist before the index is used. An index entry that is out of range will not produce some padded value from a temporarily enlarged `S`; again, it will produce an error.

4.3.2 Targets

The indexing of targets has many aspects in common with the indexing of sources, but also many differences. The purpose of indexing a target `T`, as in `T(J,K)=R`, is to add, alter, or delete parts of `T`. Indexed `T` is usually altered, sometimes only in value, but sometimes in size and shape as well. Therefore, we must distinguish between two shapes of `T`: the initial `T` (just before the assignment) and the resulting or final `T`. At times, initial indexed `T` may be nonexistent, and legally so. At those times, MATLAB creates `T`, assigns it an (usually temporary) empty value, then proceeds with the requested assignment. An index entry can often refer to a part of `T` that has not been created yet. If this reference creates a larger `T`, any new elements of `T` that have unspecified value are padded with default elements: zeros for numerical data, ASCII zeros for character data (which print as white space), and empty arrays for cell and structure data. The ultimate fate of an indexed target depends not only on the target and its indices, but also on the intermediate `R` waiting to be stored in `T`. This intermediate will influence the values in `T`, the size and shape of `T`, and the very legality of the assignment statement. Empty R is used to delete parts of `T`, while non-empty `R` is used to alter existing elements and augment `T`.

Indexing a target is often the most efficient way to alter it. For example, let `v=1:n` where `n` is large. Suppose the nth element of `v` is to be changed from `n` to `x`. Either of the following two statements will do the job:

```
v=[v(1:n-1),x];        v(n)=x;
```

In the first statement, where `v` is indexed as a source but not as a target, almost an entire new copy of `v` is required to form the square-bracket expression. By contrast, in the second statement, where only the target `v` is indexed, only the element `v(n)` is used.

4.4 Single *versus* double indices

A source or target may be singly-indexed, as in `S(J)`, or doubly-indexed, as in `S(J,K)`. For the same `S`, `J` in `S(J)` can have a much different meaning than in `S(J,K)` or `S(K,J)`. The shape of a single index can infuence the shape of the result, whereas multiple indices are always used in vecked form. In particular, when a source `S` is singly indexed, as in `S(M)`, the resulting intermediate has the shape of M unless both `S` and `M` are vectors.

4.4.1 Single Indices

Single indices usually refer to elements in `vec(S)` or `vec(T)`:

```
S=[1,2,3; 4,5,6];   S([2,5]) -> [4; 3]
```

Note that the expression `S([2,5])` contains a single index, namely the vector `[2,5]`. That vector is a single entity in MATLAB, for indexing as well as other purposes. Therefore, single-index rules apply. The expression `S([2,5])`, which has one vector index, is not to be confused with the expression `S(2,5)`, which has two scalar indices.

4.4.2 Double Indices

The first of a pair of double indices refers to the rows of `S` or `T`. The second of the pair refers to columns. The first index may be of a different type than the second (see next section about types), e.g., an ordinal first index and a logical second index, as in:

```
S=[1,3,5; 2,4,6];   S([1,2,1],[1,0,1]>0) -> [1,5; 2,6; 1,5]
```

Here, the first index vector `[1,2.1]` says "copy rows 1, 2, and 1 of `S` into `R` in that order", whereas the second index vector `[1,0,1]>0`, being a logical index, says "copy columns 1 and 3 of `S` into `R`, omitting column 2".

4.5 Types of indices

Indices are of four types: empty, logical, ordinal, and default.

4.5.1 Empty indices and intermediates

- Empty indices in sources produce empty intermediates. If multiple indices are used on a source, and one of them is empty, the dimension of the intermediate may still have have some non-zero dimensions even though the intermediate is empty. For example, if size(S) is [3,4,5], a 3-D array, then S(:,[],:) will have size [3,0,5], and will be empty.
- Empty single indices can be used in targets as follows: `T(J)=R` where both `J` and `R` are empty, all of which leaves `T` unchanged.

The empty intermediate `[]` is used to delete parts of the target `T`:

```
T = [ 1,2,3; 4,5,6; 7,8,9 ];   % T is 3-by-3.
T( :, 2 ) = [];   % Column 2 is gone, T is 3 by 2.
```

For more about this feature, see the subsection on default indices below.

4.5.2 Logical indices

For an index to be classed as logical, it must be created by one of the following methods:

- Conversion from numeric data, e.g. `K=logical(A)`, where `A` is a matrix of numbers. `K` will be a copy of `A`, but its use in a logical context will depend only on whether the numbers are non-zero (for "yes" or "true") or zero (for "no" or "false").
- Use of the relational operators `( ==, ~=, <, <=, >, >= )`, e.g. `K = A<B;` where `A` and `B` are matrices of arbitrary numbers and `K <-` a matrix of 1's and 0's.
- Use of logical operators or functions:

```
and(&), or(|), not(~), xor, any, all, exist,
isempty, isinf, isNaN, issparse, isa, etc.
```

- Use of array operations (subscripting, square-bracket construction, and the `cat` function) on existing logical arrays.

There are two flavors of logical variable: strictly zero-one, and the less restrictive zero-nonzero type. You should be aware of which type you are using. As an example of the distinction, consider two codes that attempt to count nonzero elements in a vector:

```
Right code:   K = (v~=0);       nright = sum(K);
Wrong code:   K = logical(v);   nwrong = sum(K);
%   nwrong  <-  sum(v).
```

Zero-nonzero logical arrays are useful when the numbers have both numeric and logical meanings, and you don't want to create redundant 1-0 arrays just to do the logic. For simplicity, we will use zero-one logic in the rest of the discussion, converting numeric zero-one expressions to logical e.g. by the operation `[1,0]>0`.

In a source index, logical 1 means "include", while logical 0 means "exclude" or "omit":

```
S = [1,2,3; 4,5,6; 7,8,9];   J = [1,0,1]>0;   K = [0,1,1]>0;
% Result:   S(J,K) -> [2,3; 8,9].
```

In a target index, a logical 1 means "put incoming data here", while a logical 0 means "leave this part as it is":

```
T=zeros(3,3); T([1,0,1]>0,[0,1,1]>0) = [5,6; 7,8];
% Result:   T <- [0,5,6; 0,0,0; 0,7,8]
```

For either sources or targets, it is the *positions* of the ones in a logical index that determine which parts of a matrix will be accessed or altered.

4.5.3 Ordinal indices

If an index `K` does not meet the requirements of a logical index, it is regarded as ordinal, i.e., an index in the usual numeric sense. For sources, the value of an index entry tells which part (element, row, or column) of the source is to be accessed, while the position of an ordinal entry tells which part of the result those elements will be stored in:

```
S = [3,6,9];   % S([1,3]) -> [3,9]
```

For targets, the value of an index entry tells which part of `T` is to be altered, while the position of that entry tells which part of `R` will be stored there:

```
T = zeros(1,3);   T(2:3) = [5,6];   % T <- [0,5,6].
```

Ordinal indices are far more powerful than logical indices. For any logical index `L` with at least one 1 entry, there is an ordinal index `K` that will do the same job. E.g., the second and third statements below are equivalent:

```
S = rand(1,6);   T = S([0,0,0,1,1,1]>0);   T = S([4,5,6]);
```

That is, ordinal indices can cause inclusions and omissions, just as logical indices can. In addition, ordinal indices can specify permutations and repeats, in principle without limit:

```
S = [3,6,9];   % S([1,2,3,2,1]) -> [3,6,9,6,3]
```

Further, when used singly, ordinal indices can dictate the shape of the result, e.g.:

```
S = [3,6,9];   % S([ 1,2; 3,1 ]) -> [ 3,6; 9,3 ].
```

4.5.4 Converting logical to ordinal

An ordinal index may have no logical equivalent, especially if it implies reorderings and repeats. But it is always possible to convert a logical index to an equivalent ordinal form. The MATLAB-supplied `find` function will do the job:

```
L = [1,0,0,1,0,0]>0;  f = find(L);   % f <- [1,4];
```

The logical form is often generated first, because it is the most convenient way to produce the results of a logical test on a vector, e.g. `v>-1`. However, the ordinal form is often more efficient for subsequent processing, because it may contain far fewer elements.

4.5.5 Default indices and cross-section removal

The default index, denoted ":", acts as the column vector of all integers running from 1 to some contextual maximum value. E.g.:

```
S=[1,3,5; 2,4,6];   S([1,2,1],:) -> [1,3,5; 2,4,6; 1,3,5];
```

In this context, ":" stands for the index vector `[1,2,3]`. Some rules for determining default indices are:

- A single default index ":" runs from 1 to the number of elements in `S` or `T`, i.e. `1:length(vec(S))` or `1:prod(size(S))`.

  ```
  S=[2,4; 6,8];   S(:) -> [2; 6; 4; 8]
  ```

- For doubly-indexed sources, ":" as a first index runs from 1 to the number of rows in the item, while ":" as a second index runs from 1 to the number of columns.

  ```
  S = [  1,2,3;  4,5,6;  7,8,9  ];
  S( :, 2 )  ->  [ 2; 5; 8 ];
  S( 2:3, : )  ->  [  4,5,6;  7,8,9  ]
  ```

Default indices have a special role in deleting parts of arrays. Cross-sections (rows, columns, pages) of an array can be deleted by storing an empty array in them:

```
T = rand(10,10,10);   % T has 10 rows, columns, and pages.
K = 5:7;              % Index K is [5,6,7].
T(:,:,K) = [];    % Pages 5,6,7 are gone; T has 7 pages.
```

Note in the last line that the target `T` has only one non-colon index, and this is the general rule when storing empty information. The restriction to one non-colon index prevents such errors as `M(2,3)=[]` where `M` is a 3-by-3 matrix. It is uncertain what kind of result one would want from the deletion of a single matrix element, or any non-cross-sectional subset of elements.

Targets can become empty by removing all cross-sections of a specific type. In the above example, if `K` had been `1:10`, then all of `T`'s pages would be deleted, leaving `T` empty. However, one property of the default index is that it does not zero out the corresponding index in such a case. That is, the size of `T` would be `[10,10,0]`, not `[0,0]` as one might expect. Nonzero dimensions of an empty array are of use when conformality is required, e.g. in matrix multiplication. For example, the operation `A*v`, where A is 3 by 3, requires that `v` have 3 rows. Even if `v` is empty, it must have size `[3,0]` (not `[0,0]`), whereupon the product will likewise have size `[3,0]`. Proper use of default indices when creating empty arrays will maintain conformality.

4.6 Case in point: vectors

Single indices hold a special place in the MATLAB indexing scheme. MATLAB even provides functions `ind2sub` and `sub2ind` to convert single indices to and from multiple subscripts. Since vectors are usually singly indexed, they become a special topic. Also, because vectors are the simplest kind of multi-element array, they are used often, taking on an importance worthy of separate discussion.

Vectors with two or more elements can have either row or column orientation. Appending elements by target single indexing preserves the row-column nature of the target:

```
V = [10,20];   V([3;4]) = [30;40];   % V <- [10,20,30,40]
V = [10;20];   V([3,4]) = [30,40];   % V <- [10;20;30;40]
```

Notice that the row-column orientation of the target prevails over the orientations of indices and intermediates.

Difficulties can arise when the vector is first defined, or temporarily cleared, or when it has been reduced to a scalar or empty by storing empty information in it:

```
clear V;         V(1) = 50;     % V <- 50, a scalar.
V = 10:10:90;    V(2:9) = [];   % V <- 10, a scalar.
V = 10:10:90;    V(1:9) = [];   % V <- [].
```

at which time `V` has neither row nor column orientation. Subsequent extension of `V` by target single indexing will result in a row, regardless of any previous orientation:

```
V=[3;4];   V(2)=[];   V(2)=4;   % V <- [3,4], not [3;4].
```

Also recall that colon notation produces a row, so that initializing `V` as `V=1:9` will produce a row.

While row orientation is easy to initialize and maintain, column orientation is often preferable because it may be expected by the matrix operations you are using. If a vector `V` is frequently accessed as a column, it may still be stored as a row, and referred to as `V(:)` whenever column orientation is required, but this requires the generation of an intermediate vector with column orientation. Hence, you may prefer to keep `V` as a column, which is easy if `length(V)` is fixed. For varying `length(V)`, which may be 1 or 0 at times (initialization being a frequent case), here are some ways to maintain column orientation:

```
% Code 1: For the case where index K is an arbitrary array.
if isempty(X);           % If X is empty, use a single index to
   V(K) = [];            %    delete the Kth elements of V.
else                     % Otherwise, use vec(X) and double
   V(K,1) = X(:); end    %    indices on V to ensure column.
% ------------------------------------------------------------
% Code 2: To add to or delete from the end of V,
%            one element at a time.
V = zeros(999,1);   % Preallocate maximum space for V.
n = 0;              % Initialize nominal size of V.
n=n+1; V(n)=x;      % Add an (n+1)st element to V.
n=n-1;              % Delete nth element (n~=0).
% ------------------------------------------------------------
% Code 3: Same purpose as code 2.
V = zeros(0,1);        % Initial V is a empty column.
V = [ V; x ];          % Add an element to column V.
V(length(V)) = [];     % Delete last element from V.
```

Code 1 is versatile, enabling alteration of existing elements, as well as addition and deletion. Its disadvantage is that, should the size of `V` fluctuate widely, much time might be spent looking for workspace to store `V` at its peak sizes. For some applications, it may be best to preallocate space for `V`, as in Code 2. This strategy obliges the user to keep track of the number of elements `n`, but with sufficiently large initial space, `V` never needs overhead to find more space. On the face of it, Code 3 looks simpler than Code 2, but as in Code 1, `V` can grow, demanding more and more space. Further, unlike Codes 1 and 2, whenever an element is added to `V`, a copy of `V` is generated to form `[ V, x ]`, which compounds the overhead problem. Such overhead considerations become serious with intense repetition and/or large magnitude and fluctuation of `length(V)`.

4.7 Thinking in many dimensions

An n-dimensional array of numbers is indexed `A(I1,I2,...,In)`, where the jth index `Ij` may be of any type described in the previous chapters. The first two indices are like the indices of a matrix, describing the row and column positions of the elements. If you think of each matrix (block of columns) residing on a separate page, then the third index tells which page we are talking about. If you think of each block of pages residing in a separate book, then the fourth index tells which book we are talking about, and so on through shelves, walls, rooms, buildings, cities, counties, states, nations, planets, solar systems, galaxies, universes, and whatever you think lies beyond that.

4.8 Storage and addressing

No matter how many indices you use, the storage of the numbers is much the same as for a vector. Recall that a matrix is stored one column vector after another as we proceed across the page. If there is more than one page, the pages will likewise be stored one after the other, and so on. For example, the storage of a 2-by-3-by-4 array is indistinguishable from that of a 24-vector. Vectors are distinguished from 3-D arrays because, off in separate entries somewhere, MATLAB records the dimensions of each array. This uniformity of storage enables the user to address any array as though it were an array of fewer dimensions, just as a matrix can be addressed as a vector. For example, a book can be addressed as a matrix or a vector, so that the various styles of indexing are:

```
% Let i, j, and k scalar indices.
% Let A be D1-by-D2-by-D3.
z = A(i,j,k);   % z  <-  A row i col j page k.
z = A(i,j);     % z  <-  A row i col j, where A
                %    is treated as a matrix with
                %    D1 rows and D2*D3 columns.
Z = A(i);       % z  <-  A row i, where A is treated
                %    as a vector with D1*D2*D3 rows.
```

Think of indexing a book (3-D) in matrix (2-D) style as tearing all the pages out of the book and taping them side by side, making a single large page or matrix. Likewise indexing a shelf (4-D) like a book (3-D) is like taking all the books on a shelf, ripping off all their covers, and binding them as one huge book. The analogy is not exact because books on a real bookshelf are not obliged to have the same number of pages, nor are the pages obliged to have the same number of rows and columns. But in arrays, strict uniformity is the rule.

4.9 Efficiency

The more dimensions you use for indexing, the more expensive it is to compute the actual location of the desired element in the storage vector. For example, let `Z` be a 4-D array with dimensions `D1, D2, D3,` and `D4`. To address the element `Z(i1,i2,i3,i4)`, where the indices are scalars, one must locate that element in the `Z` storage vector by computing a single vector index `j` from the four original i's:

```
j = i1 + D1*(i2-1) + D1*D2*(i3-1) + D1*D2*D3*(i4-1)
  = i1 + D1*( i2-1 + D2*( i3-1 + D3*( i4-1 )))
```

This formula can be avoided in certain common cases, e.g. when accessing successive elements of a row or some other cross section, but in the case of random access, indexing effort can easily dominate computation effort if one is cavalier about the use of multidimensions. Using array operations, as opposed to scalar operations in loops, will likely cut down on the burden of dimension. Also, where possible, index the array with as few dimensions as possible. To this end, MATLAB provides a function that computes the vector (single) index of an element of an array of arbitrary dimension and size. The following code does the job for a 4-D array:

```
% Let i1, i2, i3, and i4 be scalar indices.
sz  = size(A);   % sz is a 4-vector of size.
Ind = sub2ind(sz,i1,i2,i3,i4);
```

If the element `A(i1,i2,i3,i4)` is to be accessed repeatedly, it may now be addressed as the vector element `A(Ind)` without the 4-D burden of indexing.

The function `sub2ind` is also of use when arbitrary parts of an array are to be accessed. As an elementary example, consider the problem of replacing the main diagonal of a large n-by-n matrix `A` with an n-vector `v`:

```
Code 1:   for i=1:n;   A(i,i)=v(i);   end;
Code 2:   A = A - diag(diag(A)) + diag(v);
Code 3:   Ind = sub2ind( [n,n], 1:n, 1:n );
          A(Ind) = v;
```

Code 1 uses a loop and double indexing. Code 2 avoids loops, but at the cost of creating large intermediate matrices. Code 3 avoids loops, double indexing, and the creation of large matrices, while accessing only those parts of `A` that need changing. Further, the first line of code 3 need be executed only once, no matter how many times the diagonal of `A` is changed (provided `A` doesn't change size).

4.10 Dimensionality

MATLAB provides functions that access the dimensions of an array. The expression `size(A)` will return a vector of dimensions, not including trailing singletons. The expression `ndims(A)` will return the number of dimensions of A, again, not counting trailing singletons, i.e. `ndims(A)` is equivalent to `length(size(A))`. However, the number of dimensions is always 2 or greater, even if `A` is vector, scalar, or empty. Here are some examples:

```
Definition of A        size(A)     ndims(A)
 A = [];               [0,0]       2
 A = 5;                [1,1]       2
 A = [4,6];            [1,2]       2
 A = zeros(1,1,4);     [1,1,4]     3
 A = zeros(1,4,1);     [1,4]       2
 clear A;
 A(2,1,1,2,1) = 7;     [2,1,1,2]   4
```

4.11 Scalar functions of vectors

MATLAB provides several functions that produce scalar results from vector input, namely:

```
all any max min mean median prod std sum
```

The function `sum` will illustrate how these built-in functions operate in higher dimensions. If `v` is a vector, the expression `sum(v)` produces the sum of the elements of that vector. If `A` is a higher-dimensional array, then `sum(A)` produces the sum(s) along the first non-singleton dimension of `A`, producing an array in which the first non-singleton dimension of `A` is replaced by a singleton. For example:

```
Definition of A     size(A)      size(sum(A))
A=rand(3,5);        [3,5]        [1,5]
A=[1,2,3];          [1,3]        [1,1]
A=rand(1,3,5);      [1,3,5]      [1,1,5]
A=rand(1,1,1,7);    [1,1,1,7]    [1,1]
```

These functions accept an optional second argument (or third argument in the case of max and min. See Help.) specifying the dimension over which the function is to operate. Thus the expression `sum(A,2)` will produce the row sums of a matrix `A` rather the column sums, even if there is only one column. The two-argument form of these functions gives you absolute control over which dimension is operated upon. The result is an array in which the specified dimension is reduced to a singleton, while the other dimensions are the same as those of the input array. For example:

```
                              A = rand(5,5,5);
M = rand(5,5);                size(sum(A,1)) is [1,5,5]
size(sum(M,1)) is [1,5]       size(sum(A,2)) is [5,1,5]
size(sum(M,2)) is [5,1]       size(sum(A,3)) is [5,5]
```

4.12 Functions for multidimensionality

MATLAB provides several functions to assist you in various tasks associated with multidimensional arrays:

```
cat ind2sub ipermute length ndgrid ndims
permute reshape shiftdim size squeeze sub2ind
```

Some of these have been previously mentioned.

The function `cat` provides a neat way to compose multidimensional arrays. For example, let `M1, M2, M3,` and `M4` be matrices of like size that we wish to join as pages of a 3-D array. The statement `A=cat(3,M1,M2,M3,M4)` does the job. The first argument to `cat` is an integer denoting the dimension along which the joining occurs.

The function `squeeze` squeezes out singleton dimensions, so that, e.g.:

```
A = rand(1,3,1,5);    % size(A) is [1,3,1,5]
B = squeeze(A);       % size(B) is [3,5]
```

Details about the functions `cat`, `squeeze`, and the others can be found in the help files by saying e.g. `help cat`.

4.13 Array types

Numeric arrays, which we have been discussing in some detail, comprise only one of several array classes. There are also character string arrays, cell arrays, structure arrays, and object arrays. The syntax of indexing is similar for all of these, although the elements of cell arrays and structure arrays can differ wildly in both size and class. For example, an element of a cell array might be another cell array, while a neighboring element might be a single number. Some of the functions that aid in manipulating numerical arrays e.g. `cat` and `squeeze`, are also capable of the other classes of arrays. In addition, of course, each class of array has its own collection of specialized functions.

4.14 Empty arrays

Yes, you can have multidimensional empty arrays, and in a variety of data classes as well. Regardless of class, a statement of the form:

```
% x has size [ D1, D2, ... Di, ... Dn ]
x( :, :, ... :, 1:Di, :, ... ;, ; ) = []
% x has size [ D1, D2, ...  0, ... Dn ]
```

renders `x` empty, having zeroed out the ith dimension. However, several things about `x` are not forgotten. For one thing, all dimensions except the one that was zeroed are retained for use in building arrays by catenation. Currently, MATLAB will allow you to catenate an empty array with another array of apparently incompatible dimensions (an empty should fit with anything, right? Wrong!), but it will give a warning message, and presumably in a future MATLAB, such a construct will be illegal. The number of dimensions remembered can be reduced by using fewer default subscripts. In the above example, if `m=n-1`, using `m` indices instead of `n` produces an empty array of size `[D1,D2,...0,...,Dm*Dn]`.

In addition to past dimensions, the class of the empty `x` array is also remembered, and in the case of empty structure arrays, the field names are remembered. **Exception:** if a structure array is emptied by removing all of its fields, using the `rmfield` function, the resulting empty array is of class `double`, not `structure`, and its size is `[0,0]`.

Chapter 5

Strings

Along with the floating point number, the character is a fundamental datum in MATLAB. All told, there are over 30 built-in functions for handling characters, including tests for various types of character data, conversion from character data to numeric and back, execution of strings as MATLAB expressions, and operations on strings, such as concatenation and substring comparison. These may be found in the chapter on strings in "Using MATLAB", or in the help files under `strfun`.

5.1 The ASCII character set

Each character is stored as a 2-byte ASCII code, namely one of the integers from 0 to 255, which are displayed in character form. To see the ASCII code for a character, convert the character to double, e.g. `double('A')`, which produces the numeric value 65. Inversely, the integers 0 to 255 can be converted to character, e.g. `char(50)` produces the character `'2'` (as opposed to the number 2). The ASCII character set includes the upper case alphabet (codes 65 to 90), the lower case alphabet (codes 97 to 122), and the numerals 0 to 9 (codes 48 to 57). Also included are alphabetic characters with diacritical marks, mathematical operators, printer controls, and various special characters.

5.2 Out-of-range codes

Note that the 2-byte character format accomodates the integers from 0 to 65535, any of which will be accepted without complaint, but the character displayed corresponds to the remainder `rem(c,256)`, i.e. the code in question modulo 256. In other words, the high-order byte is ignored, and no out-of-range warning is given. The integer code itself will *not* be truncated to the low order byte. Perhaps MathWorks, Inc. has future plans for the codes 256 to 65535, so it is best not to rely on their current status. Currently, `char` will also accept complex numbers without complaint, ignoring the imaginary part, as long as the real part is within 2-byte range. The `char` function will convert all other numeric values (whether those values be huge, fractional, or negative) to 2-byte form with a warning message.

5.3 Characters as integers

Character variables and characters *per se* may appear as quantities in mathematical expressions, such as:

```
c1=char(100);    % 100 is the code for 'd'.
c2=char(c1+1);   % 101 is the code for 'e'.
```

Characters in mathematical expressions like `c1+1` are converted to the double-precision equivalents of their ASCII codes, and the expression is evaluated in the usual 8-byte format. The result of the expression can then be stored in 2-byte format using the `char` function as above. Of course, the results should always be in the integer range 0 to 65535, or with an offset value, you can interpret some of them as negative integers, e.g. -32768 to +32767. So, effectively, a character variable can be treated as a short integer, providing a way to store large arrays of integers in a compact format. However, the use of a character as an index is not allowed. You must convert the character to double first.

5.4 Character arrays

Characters usually occur in rows called strings, which in turn can be in higher dimensional arrays. An explicit string is always bracketed by single quotes:

```
' % between quotes does not indicate a comment.'
```

If the above line appears in a function, its execution causes the line to be printed, just like the result of any expression. The statement:

```
z='zoom';
```

defines the variable `z` as a string of four characters. A matrix of characters is also legal:

```
element_type = [ 'real   ' ; 'complex' ; 'string ' ];
```

produces a 3-by-7 matrix of characters, usually thought of as a series of 3 strings. In many ways, manipulation of character arrays resembles the manipulation of numeric arrays. The above example is analogous to the formation of a 3-by-7 numeric matrix using square-bracket notation. The indexing conventions are the same, as is the notation for building larger arrays from smaller ones.

5.5 Strings that represent numbers

A string can be converted to a number if it represents one, and a number can be converted to a variety of string formats. For example, the number `pi` can be converted to any of the strings:

```
'3'   '3.14'   '3.14159265'   ' 3.14e00'   etc.
```

and strings of the above type can be converted to numbers. The process is simple, but with many options that we will not catalog here. Consult the help file under `strfun` and the references for details.

5.6 The eval function

A single string can be executed as MATLAB code using the `eval` function. If the string contains a MATLAB expression, `eval` returns the value of that expression. However, the name `eval` is an understatement, because `eval` is not restricted to evaluating expressions. The string argument can contain any long sequence of MATLAB statements separated by semicolons. (MATLAB strings can be very long.) These statements may include `if` statements, loops, function calls, and even commands to the operating system using `!` as the initial character. Additionally, the `eval` function may appear within loops, so it is quite possible to execute entire programs using `eval`.

Be aware, though, that executing strings can affect efficiency. In the following example, four loops were timed using the MATLAB etime function as in the chapter on time and work:

```
% Time for four deliberately slow loops.
t=0:.006:6;  str1='sin';  str2='sin(x)';  str3='feval(str1,x)';
for x=t,  y = sin(x);          end     % 1.15 seconds.
for x=t,  y = feval(str1,x);   end     % 1.64 seconds.
for x=t,  y = eval(str2);      end     % 3.96 seconds.
for x=t,  y = eval(str3);      end     % 5.61 seconds.
```

The degradation of efficiency seen above is really a worst case, because `sin(x)` itself takes hardly any time. (It registers as 1 flop in the MATLAB flop count.) When eval and feval (discussed below) are invoked on a costly function, e.g. `inv(x)` for large `x`, their overhead time may hardly be noticed.

5.7 File-tending applications

In essence, `eval(str)` simply does whatever MATLAB would do if the string str appeared as code, only slower. One housekeeping use of this feature is to manipulate files in an orderly way:

```
% Collect the vectors stored in files zap01.mat
% through zap99.mat, and store them in the matrix ZAP.
% Clear each vector before loading the next to avoid clutter.
% The for-loop below composes strings for eval.
% E.g., when j=6, the string becomes:
% 'load zap06; ZAP(:,6)=zap06; clear zap06'.
sload='load ';  szap='zap';  szap0='zap0';  sZAP='; ZAP(:,';
seq=')=';  sclear='; clear ';
for j=1:99,  sj=int2str(j);
    if j<10,    sfile = [szap0,sj];
    else        sfile = [szap, sj];   end;
    eval([sload,sfile,sZAP,sj,seq,sfile,sclear,sfile]);
    end;
```

It is not necessary for the files to have such orderly names. You can keep chaotic file names in string variables with orderly names, say zap01 through zap99. (Note, zap01 through zap99 are *variable* names, not file names.) In the next example, the only change to the previous example is in the `if-else` statement:

```
% convert string names to file names.
if j<10,   sfile = eval([szap0,sj]);
else       sfile = eval([szap, sj]);   end;
```

You can also issue a series of commands to the operating system that might otherwise have been very time-consuming. E.g., you have written 50 chapters of a book, which are in 50 separate files called CHAP01 through CHAP50. You now wish to collect those files into a large master file called BOOK, and email it to your publisher. This collection problem is analogous to the previous two examples, but now the generated strings will contain operating system commands:

```
'!COPY   BOOK + CHAP06   BOOK'
```

Of course, the chapters themselves can have chaotic file names, but how to revise the above example to accomodate this can now be left to the reader.

5.8 The feval function

If you try to invoke a function called `func` in a code where there is a variable named `func`, the variable will take precedence, and the function will be ignored. The arguments to the function will be treated, mistakenly, as indices of the variable. When you have a choice of names, it is easy to keep distinct names for your variables and your functions. But when you are writing a function that accepts another function name as a string argument, you may not have control over which name the user will put in that string:

```
function Y = F(S,X);  % S is a string.
Z = X^2;              % X is a scalar.
Y = eval([S,'(Z)']);  % Treat S as function.
```

In this example, the argument to `eval` is the string in `S` followed by the string `'(Z)'`. For example, if `S` contains `'func'`, then the argument to `eval` becomes `'func(Z)'`, and everything works as expected, because the name `'func'` is not used elsewhere in function `F`. However, if `S` contains `'X'` or `'Z'`, which are variable names in the function `F`, the attempt by `eval` to invoke the function `X` or `Z` fails. To eliminate this possibility, the `feval` function is provided. Replacing the last line of the above example with:

```
Y = feval(S,Z);
```

forces the string in `S` to be interpreted as a function name. The remaining arguments to `feval` become the arguments to that function. This is the preferred way to treat a string as a function name.

5.9 Parsing strings

Interpretation of strings is not limited to MATLAB entities like expressions, statements, and function names. Within MATLAB, you can develop your own language, using MATLAB's string-handling features to help interpret the statements. A useful string function for this purpose is `strtok`. Let's say the elements of your language are 1) variable names of one or more alphanumeric characters, e.g. `pH7`, 2) Numeric constants which always begin with a numeral, 3) operators like + and *, and 4) separators like commas and blanks. The `strtok` function can help to decompose the string into its syntactic tokens. The first line of the code:

```
[ T, R ] = strtok( S, D );
if isletter(T(1)), ... % T is a variable name.
else ...               % T is a numeric constant.
```

will scan the string `S` for any substring delimited by the characters in the delimiter string `D`. The delimited substring, called a token, is returned in the output `T`, and the remainder of `S` is returned in R for further scanning. If, e.g,, you are scanning for a variable name, then the delimiter string `D` might contain all the non-alphanumeric characters in your language. Finally, the `if` statement above will decide if the token is a variable name or a numeric constant.

Another built-in string function, `strrep`, replaces one substring with another:

```
S1 = 'x+x^2+y-x1';
S2 = strrep(S1,'x','z');    % S2 <- 'z+z^2+y-z1'.
```

where every instance of `'x'` in `S1` is replaced by `'z'` in `S2`. This device is especially useful in mathematical deduction and theorem-proving, but it must be used with caution, lest you replace parts of variable names by mistake. In the above example, note that `'x1'` was changed to `'z1'`, which may have been unintended.

5.10 Treating files as strings

The lines of an ASCII data file can be treated as strings, not necessarily of equal length. Each line can be read in turn and processed as follows:

```
fn = 'myfile';          % fn = file name.
fid = fopen(fn,'r');    % Open fn as 'read only'.
while 1;                % Loop 'forever'.
   s = fgetl(fid);      % Read a line into s.
   if ~isstr(s);        % If end of file,
      break;   end;     %   leave the loop.
   % ----- Process string s here. -----------
end;
```

5.11 Strings in cells

Character arrays contain strings of uniform length. A character array is much like a numeric array, in that every row must have the same number of characters. Shorter strings in the array must be padded with blanks to conform. Often, this requirement is an inconvenience that can be avoided by using cell arrays, which have no restrictions on the sizes of entries:

```
citizen{1,ncit} = 'John Q. Public';
citizen{2,ncit} = '1234 West Main Lane';
citizen{3,ncit} = 'Townsville MD 56789-1234'; ... etc.
```

For some applications, it is convenient to use both character-array and cell-array formats. To facilitate this, MATLAB provides conversion functions:

```
B = cellstr(A);    % Convert character array A to cell
                   % array B, stripping off trailing blanks.
A = char(B);       % Convert cell array B to character
                   % array A, padding with blanks where needed.
```

Chapter 6

Cells and Structures

Cell and structure arrays can contain any kind of information, including other cell and structure arrays. By nesting cell and structure arrays, one can create any tree-structure type of data organization, where meaningful data (e.g. numeric and string arrays) reside in the leaves of the tree (or in its root tips if you prefer a top-down chart). The programmer is in complete control of how the tree is branched and what kind of information is found in the leaves. This chapter tells how such information is stored and accessed.

6.1 Cells

Cell arrays can contain, in their elements, any kind of data, e.g. numerical arrays, strings, or even other cell arrays. Whenever a datum can contain its own kind in this manner, very careful use of notation is required to specify the level of depth to which one is referring. A cell array is analogous to a tool cabinet with many drawers. You can talk about the whole cabinet, a particular set of drawers, or the contents of that set of drawers. Even though a drawer is full of screw drivers, you cannot drive a screw with the drawer itself. Similarly, even though a cell contains a numerical vector, you cannot do a vector operation the cell itself. You must operate on the vector within the cell. To make the distinction, the cell itself is denoted by its name followed by parenthesized indices, e.g. `c1(i,j)`, while the content of that cell is denoted with braced indices, e.g `c1{i,j}`. Braces are also used in explicit definition of cell arrays, much as brackets are used in matrix definitions. The following example illustrates these ideas.

```
c1 = cell(2,3);      % Create a 2-by-3 cell array containing nulls.
c2 = c1(1:2,1:2);    % Cell array c2 <- a 2-by-2 sub-array of c1.
c3 = {'a', pi;...    % Define 2-by-2 cell array c3 by a braced list,
      50, 'efg'};    %   which is an explicit cell array.
c3{2,2}(3) = [];     % Remove the 'g' from the string in c3(2,2).
                     %   Note the use of braces and parentheses.
r = rand(99,99);     % r <- square random matrix.
c1(2,2) = r;         % Bad! A cell per se cannot be
                     %   replaced by a matrix.
c1{2,2} = r;         % Good! The content of a cell may be
                     %   may be replaced by a matrix.
c1{1,4} = c1;        % A complete copy of cell array c1 is placed
                     %   within cell c1(1,4). Because c1 was only
                     %   2-by-3, it has been expanded to 2-by-4.
                     %   c1(2,4) now contains an empty matrix.
c1{1,4}{2,2}(7,9)    % Display the (7,9) element of the 99-by-99
                     %   matrix contained in the (2,2) element of
                     %   the (unnamed) cell array which in turn
                     %   resides in the (1,4) element of c1.
c1(:,4) = [];        % Delete the 4th column of c1.  This is a
                     %   special syntax for cross-section removal.
                     %   It does not put nulls in place of cells.
c1{1,2} = [];        % Replace the contents of c1(1,2) and
c1{2,2} = [];        %   c1(2,2) with nulls.  This does not
                     %   remove the 2nd column of cells from c1.
c1{2,3} = {};        % Replace the contents of c1(2,3) with an
                     %   empty list, which is not an empty matrix.
```

To carry the tool cabinet analogy to its unnatural extreme, we can imagine that some drawers can actually contain smaller tool cabinets and so on, like nested Russian dolls. Denoting a particular tool from such nested drawers requires a pair of braces for each level of nesting, followed by a parenthesized index specifying the tool. Further, we can imagine a telescoping cabinet design, so that when a row or column of drawers is removed, the cabinet shrinks to conform. Ah, analogies!

A cell array is a (possibly multidimensional) rectangular array, whose size and access to elements are governed by the same rules and prohibitions that govern numerical arrays. However, the distinction between a cell and its content occasionally causes some indexing confusion. Given a cell array `C` of size 2-by-3, storing other dimensioned cell arrays in existing elements of `C` does not change the size of `C` itself:

```
C = cell(2,3);
C{1,3}= cell(1,2);
```

The second statement above does not expand `C` from 2-by-3 to 2-by-4. Rather, it simply places a 1-by-2 cell array into the element C(1,3). The size of `C` remains 2-by-3, unaffected by the size of the content of its elements. To expand `C` to 2-by-4, one might use e.g. any one of the following six statements:

```
C(1,4)=C(1,1);   C(1,4)={[]};   C(:,4)=C(:,1);   % line 1
C{1,4}=C{1,1};   C{1,4}= [] ;                    % line 2
[ C{:,4} ] = deal(C{:,1});                       % line 3
```

In line 1, a cell is defined by another cell, whereas in lines 2 and 3, cells are defined by content. Line 3 is a syntax whereby the contents from one cell array (or sub-array) can be transferred, cell by cell, to another cell array with an equal number number of elements.

6.2 Structures

The structure, like the cell array, is a high level of data organization, in that it can contain a mix of all other kinds of data, and may even contain other structures and cell arrays. A structure contains a set of data items, each item being described by a field name. The item is contained in the corresponding field, just as it could be contained in a cell. For example, the structure `customer` can have fields called `name`, `address`, `telno`, `faxno`, `email`, and `order`. The code for creating a new customer in an existing array of customers might be:

```
customer(n).name = 'O. Leo Leahy'
customer(n).address(1,:) = '123 Mauna Loa Drive';
customer(n).address(2,:) = 'Honolulu HI 99999  ';
customer(n).telno = '(808)555-1234';
customer(n).faxno = '(808)555-2345';
customer(n).email = 'OLL18x@uhi.edu';
customer(n).order(1).title = 'Webster's II';
customer(n).order(1).ISBN = '0-395-33957-X';
customer(n).order(1).costper = 20.95;
customer(n).order(1).number = 5;
customer(n).order(1).status = 'received';
customer(n).order(1).paydate = '1998Jan02';
```

The fields `name`, `telno`, `faxno`, and `email` contain simple character strings. The field `address` is a 2-dimensional character array that can have as many lines as needed for each customer. The field `order` is itself a structure array with six fields, four of which contain strings, while the other two contain numbers. The next order will be referred to as `customer(n).order(2)`.

The structure itself may be indexed, e.g., when there is more than one customer, as in `customer(217)`. Similarly, the fields can be indexed, e.g., when there is more than one order per customer, as in `customer(9).order(3)`. When a structure is indexed, every element of the structure array must have the same number of fields and the same field names. However, as in cell arrays, there is no requirement to have similar content in any of those fields. For example, if customer 217 is a big spender, `customer(217).order` might be a large structure array, whereas if customer 218 has just subscribed without buying anything yet, `customer(218).order` might be an empty matrix.

Often, there is a choice about what should be indexed. Consider an example in least-squares curve-fitting, where there are `n` observations, each with six fields:

```
% obs.indvar  = the independent variable, time.
% obs.depvar  = the dependent variable, concentration.
% obs.weight  = the statistical weight, reciprocal of variance.
% obs.compvar = the computed variable to compare to obs.depvar.
% obs.resid   = the weighted residual given by:
%               sqrt(obs.weight)*(obs.depvar-obs.compvar).
% obs.error   = standard error of obs.compvar.
```

If we expect to process hundreds of observations many times during a fitting procedure, should we index the structure or its fields? Put another way, should we have many `obs` structures, each with six scalar fields, or should we have one `obs` structure with six large array fields? This is an important consideration for efficiency. Curve-fitting uses the fields as vectors and matrices, i.e., each field is regarded as a mathematical entity. When the field is indexed, as in `obs.indvar(i)`, etc., each field is stored as an array, which is ideal for processing by numerical methods. As a simple example, the sum of squares of the residuals would be computed as:

```
SSR = sum( obs.resid .^ 2 )
```

By contrast, indexing the structure as in `obs(i).indvar` emphasizes each single observation as an organizational unit, thus breaking up the six arrays, assigning their elements to the structures `obs(i)`. Hence, to compute SSR from an array of structures, we first have to collect the residuals `obs(i).resid` from all `obs(i)` before summing:

```
SSR = sum( [ obs.resid ] .^ 2 )
```

which is a less efficient process. However, in our customer file (above), it is probably preferable to index the structure, because we want to access all of the information about a given customer as quickly as possible. We are not likely to want the sum of all of our customer telephone numbers, although with some frequency, we might want to sum what they all spent and paid, perhaps calling for a different organization for financial data.

While various parts of a cell array or structure array can be added or removed by indexing, the fields of a structure must be manipulated by name. To add a field, initialize that field in any one of the structure elements. For example, to add a new social security number field to the `customer` array, the statement:

```
customer(6).socsec = '123-45-6789';
```

will put the string '123-45-'6789' in the `socsec` field of customer 6, and empty matrices in all the other new `socsec` fields. To remove the `socsec` field, use the `rmfield` function:

```
customer = rmfield( customer, 'socsec' );
```

Note: through indexing, a strucure can become an empty structure, yet still retain its field names. However, if all fields are removed using `rmfield`, the structure loses its structure status and becomes an empty matrix.

6.3 Converting structures to cells

Structure arrays may be converted to cell arrays using the `struct2cell` function. The syntax is quite restrictive, but also simple. Recall that the structure `customer` described above has six fields: `name`, `address`, `telno`, `faxno`, `email`, and `order`. The fields within the `order` field (`title`, `ISBN`, etc.) do not count as fields of `customer`. They are fields of the substructure `order`. The structure `customer` is converted to a cell array `custcell` by:

```
custcell = struct2cell( customer );
```

If there are 6 fields and 999 customers in the structure, then `custcell` will be a 6-by-999 cell array. In general, the number of fields in the structure array becomes the first dimension of the cell array, and the other dimensions of the structure array become the succeeding dimensions of the cell array. The cell array `custcell` has no memory of the structure name `customer` or its six field names. However, the 999 cells `custcell(6,:)` now contain the 999 substructures from the `order` field, including their field names like `title` and `ISBN`, which do not have to be the same in name, number, or content from customer to customer.

6.4 Converting cells to structures

Cell arrays may be converted to structures using the `cell2struct` function, but this process has more options than `struct2cell`. Suppose we have the 6-by-999 cell array `custcell` (above), from which we want to create the structure array `customer` (above). First, we must choose the dimension of the cell array to be converted to fields. Let's choose the first dimension, namely 6. Then, we need to choose the corresponding 6 field names. (If we had chosen the second dimension,we would have needed 999 field names.) Finally, we invoke the function `cell2struct`. A sample code is:

```
% The 1st dimension of custcell will become fields.
fielddim = 1;
% Define a cell array of 6 field names.
fieldname = {'name','address','telno','faxno','email','order'};
% Convert cell array custcell to structure customer.
customer = cell2struct( custcell, fieldname, fielddim );
```

This code reverses the process of the previous section. In general, the structure will have one less dimension than the cell array from which it was converted. Its size will be the same as that of the cell array except that the dimension `fielddim` will be missing, having been converted to fields.

6.5 help entries

MATLAB provides several functions to aid in the handling of cells and structures:

Cells	Structures	Purpose
cell	struct	create
iscell	isstruct isfield	test types
cell2struct num2cell	struct2cell	convert types
celldisp cellplot	fieldnames	display forms
deal	getfield setfield	move contents
	rmfield	delete field(s)

Brief descriptions of these may be found by the command `help datatypes` or under `datatypes` in the chapter "Overview of MATLAB help".

Chapter 7

Objects

7.1 Built-in objects

Object-oriented programming allows you to speak your own private language, with newly defined classes of data and operations. MATLAB already has several built-in classes, called virtual classes because they don't quite behave the way user-defined classes do. These are discussed in the chapter "M-file programming" in "Using MATLAB", but the list there is not complete. Here is one possible organization for built-in classes of array:

```
array: any built-in object.
|  numeric: numbers.
|  |  uint8: 1-byte numbers 0-255.
|  |  double: 8-byte double-precision.
|  |  |  full: zeros are stored
|  |  |  |  real: no imaginary parts.
|  |  |  |  |  logical: nonzeros and zeros.
|  |  |  |  complex: imaginary parts.
|  |  |  sparse: zeros not stored.
|  |  |  |  realS: see real etc. above.
|  |  |  |  |  logicalS      |
|  |  |  |  complexS         |
|  char: alphameric data.
|  cell: varied class content.
|  structure: fields.
|  UserObject: user-defined.
```

The class `UserObject` is discussed in this chapter. Although the above outline conforms to the outline given in the manual, the above version adds more levels, so don't blame MathWorks for the elaborations. However, all of these names represent special categories (though not necessarily formal classes) of MATLAB data or data handling. Also, the above outline shows how intricate the relations between classes can become.

7.2 Object definition

An object is a structure (as described in the previous chapter) that is a member of a class. Every class has a class name, e.g., `Set`. To define the class of objects called `Set`, make a directory `@Set`, in which the methods (functions) for class `Set` will be stored. Note that the name of the directory (after the `@` symbol) must be the same as the name of the class. Class methods include construction of objects, display of objects, evaluation of expressions involving the objects, and conversion of fields from the class in question to other kinds of data. Some of the field names of an object are determined by the object's class methods. Other field names of an object may be inherited from other classes. Inheritance will be discussed later.

7.3 A simple example

MATLAB offers a suite of functions to implement the operations of set theory, namely:

```
intersect ismember setdiff setxor union unique
```

You will see these functions, among others, when you type `help ops` at the MATLAB prompt, or under `ops` in the chapter "Overview of MATLAB help" later in these notes. The trouble is that all the operations look like generic function calls, having none of the symbolic nicety of, say, numerical algebra. Here, we will use object-oriented programming to "algebratize" the appearance of set operations.

Sample functions for the directory `@Set` are given later in this chapter. In our `Set` example, we have only one field called `set`, so that an object `X` of class `Set` is a structure whose Set-based content resides entirely in the field `X.set`. The content can be anything one chooses. Our choice is either an empty matrix, or a row vector of positive integers, sorted, non-repeating. and not exceeding a given maximum. (This notion is more general than it seems. If you are interested in, say, sets of customers, then structures representing those customers can be put in a structure array, and the integers in our Sets can be their indices.)

Within the directory `@Set`, there must be a function called `Set` (matching the class name) whose task is to construct new objects of class `Set`. A three-statement example of such a constructor is:

```
function S = Set( X );   % Accept input array X.
S.set = X(:)';           % Store X as row in S.set.
S = class( S, 'Set' );   % Convert S to class Set.
% Note: without this last statement (above),
%    the output S would be an ordinary structure,
%    and could not be processed as an object.
```

However, such a terse constructor could easily produce objects that do not contain columns of sorted, unique integers within the proper range. The function `Set` listed in appendix 2 insures that the content of the `set` field conforms to specifications, or that an error is raised. In addition, we allow three flavors of definition: 1) by default, 2) from an existing Set, or 3) from any numerical array of positive integers:

```
A = Set;                % A.set <- null.
B = Set(A);             % If A is already a Set, either of
B = A;                  %  these copies Set A into Set B.
C = Set([4,1;1,2]);     % C.set <- [1;2;4].
```

The standard operations on objects of class `Set` are:

```
~A (not A): All integers (at or below
      some user-specified maximum)
      not contained in A.
A & B (A and B, set intersection):
      integers in A that are also in B.
A + B (A xor B, set exclusive-or):
      integers in A or B, but not in both.
A | B (A or B, set union):
      integers in A or B or both.
A - B (A minus B, set difference):
      integers in A but not in B.
```

However, the symbols we have chosen for our operations have other meanings in MATLAB. To overwrite those meanings for objects of class `Set`, we must code functions within the directory `@Set` with special function names that correspond to the operators in question, namely:

```
~: not,   &: and,   +: plus,   |: or,   -: minus
```

Codes for these five functions are given at the end of this chapter. For a list of overwritable operations and the names of functions to overwrite them, see the chapter "Overview of help" under `datatypes`, or type `help datatypes` in MATLAB. Currently, there is no way to override the precedence of these operations, which are retained from their non-overwritten counterparts, e.g. `a&b+c` is interpreted as `a&(b+c)`. Of course, the order of operations may be governed by explicit parentheses in the usual way. Note that you can redefine many operations including "=", fields, and subscripting. If you don't redefine "=", fields, and subscripting, they retain their usual meanings. For example, you can have a multidimensional array of objects.

Objects are rather private stuff. The fields generated by methods of a class cannot be accessed, except by other methods in the same class. If you want other programs to use the contents of those fields, you must provide methods within the class to convert those contents to accessible form. For example, typing the name of an object with no semicolon will not print the contents of that object, unless you code a function called `display` that allows such output. For another example, if you want to make the vectors in the `set` fields available for general processing, you should write a function called e.g. `double`, that converts an object of class `Set` to an object of class `double` (in this case, a row vector). The last section of this chapter contains sample listings of functions `display` and `double`.

So far, we have discussed the attributes of a single class of objects called `Set`. The sample directory `@Set` includes a constructor (`Set`), a displayer (`display`), a converter (`double`), and four operations (`not, and, xor, or`). However, this kind of single-class application is only beginning of object-oriented programming.

7.4 Precedence of methods

When more than one class of objects is being processed, these classes and their methods can be related in several ways. Consider the statement `z=f(x,y);` where `x` is of class `X`, `y` is of class `Y`, and functions called `f` appear in both `@X` and `@Y` directories. Which `f` will be used? If no provision is made, then the `f` in `@X` will be used because `x` is the first argument. However, you can insist that, say, `f` in `@Y` be used regardless of the argument order by the appropriate use of the `inferiorto` and `superiorto` functions. See the `help` entries for particulars.

7.5 Inheritance of attributes

An object belongs to the class that spawned it, and possibly to ancestor (parent, grandparent, etc.) classes. That is, it can contain fields from ancestor classes, and it can be manipulated by ancestor class methods. The class that spawns an object is a subclass of any ancestor classes that pertain.

A child-parent relationship is established in the child's constructor function, by using an extended form of the `class` function:

```
function b = B( f, g, ... p1, p2, ..., pn )
% --- code structure b here ---
b = class( b, 'B', p1, p2, ..., pn );
```

The last statement above converts `b` to class `B` with parent objects `p1` through `pn`. Thus `b` will have its own fields, along with inherited fields from `p1` through `pn`, which, let us say, are from classes `P1` through `Pn` respectively, each with its own directory of methods.

Once the first object of class `B` is established in a session, all other objects of class `B` must have the same attributes *in the same order*. If the function `B` above is first called with parents `p1` through `pn` (in that order) as the final arguments, a subsequent call with the parent objects rearranged is not permitted (parenthood is not commutative), nor can one substitute parents of some new class. Consistency of parent class is strictly enforced.

Another restriction on inheritance: the family tree must indeed be a tree, i.e. no closed loops, i.e. no incest. For example, a grandparent class with two descendant classes, which in turn have a common child class will produce an error message. Likewise, a child with two parents from the same class will not do anything useful. One parent's fields will be overwritten by the other parent.

The fields in object `b` are available only to methods of the class that spawned them. Methods of class `B` can access the fields explicitly defined in function `B` above, but they cannot the access the inherited fields. Likewise, methods of class `P1` cannot access the fields from class `B` (the child class), or from the other parent classes. In addition, the parent objects may have their own parents (grandparents of `b`) whose fields will likewise be passed to `b`. An object can have any number of parents, each of which can have any number of parents, and so on, each layer of ancestors adding to the fields that the object inherits. Again, access to those fields remains restricted to the class from whence they came. A descendant's methods may not operate on an ancestor object, because the descendant's fields are not present in an ancestor object. In contrast, an ancestor's methods may operate on descendant objects,

because a descendant inherits all the fields of the ancestor's class, and these fields remain accessible (only) to the ancestor's methods.

7.6 Class detection

It is often useful to know from which class or classes an object gets its fields. To help, MATLAB provides two functions. The first function, `class`, is used with one argument, namely, the object. It returns a single string containing the name of the spawning class, e.g.:

```
A='word';   B=class(A);   % B <- 'char'.
```

but it will not indicate if there are any ancestor classes. The second function, `isa`, used with two arguments:

```
isa( x, 'B' )
```

is a logical function, returning 1 if `B` is the spawning class *or any ancestor class* of `x`. Note that a seeming contradiction can arise. If object `x` has spawning class `B` and ancestor class `A`, then:

```
isa( x, 'B' )      % This expression is true.
class(x) == 'B'    % This expression is true.
isa( x, 'A' )      % This expression is true.
class(x) == 'A'    % This expression is false.
isa(x,'A') & ~class(x)=='A'   % See below.
```

The `isa` test and the `class` test are interchangeable for a spawning class, but not for an ancestor class. Therefore, the fifth expression above is true if and only if `A` is an ancestor class of `x`, providing a test for ancestorhood.

MATLAB does not provide a direct method for tracing the family tree (classes of fields) of a given object. The function `class` will reveal only the child (spawning) class of an object's family tree, while the function `isa` will confirm an ancestor only if you know the name of the ancestor class in question. If tracing the tree is important, it is up to you to provide a function for each class, so that family trees can be passed from class to class to user.

7.7 Aggregation

In addition to inheritance, MATLAB also supports aggregation, in which an object of one class is contained in a field of an object of another class. Once again, we are faced with a structure-within-structure situation, in which one must be careful to specify the level of embedding one is talking about, although the syntax is essentially that of structures. But beyond the structure syntax, we must also be careful to use the right class of methods on each embedded object because of the above restrictions on field access.

7.8 Private methods

There is a kind of cloak-and-dagger aspect to all this proprietary use of fields, an aspect that can be applied to methods as well. A class of objects can be created for which no single programmer knows everything about the rules or contents. However, if one chooses, class restrictions can be tossed aside. Class converter functions can be made freely available, so that fields of a class can be processed just about anywhere, even in other class methods. Such cross-talk defeats the purpose of object-oriented programming, which is to encourage the use of only the essential attributes of an object at each level of abstraction.

If, for either security or intellectual purity, you wish to keep some of your methods strictly within the class, you can set up a subdirectory called `private`. Such a subdirectory should not be put in a MATLAB search path. All functions in `private` are available only in `private` itself and the immediate parent directory, but not outside the parent, nor in other subdirectories of the parent. Any class directory, indeed, any directory, can have its own `private` subdirectory. Recall also that a function can be hidden by placing its code in another function, where it is available only within that function.

Within a class directory, functions in the `private` subdirectory have precedence over like-named functions elsewhere, so that the expression `sin(x)` used in a class directory will use the function `sin` from the `private` subdirectory, if a function called `sin` is in there. Outside the class directory, there is no knowledge of that hidden function, so the default `sin` (or some other local precedent) is used.

7.9 What do you gain?

As an example of object-oriented programming, consider MATLAB itself. Looking at the outline of built-in MATLAB objects at the beginning of this chapter, we see that all MATLAB objects are arrays. Arrays have certain attributes in common. For example, they can be indexed to get at their parts, and they can be attached to each other by catenation using square-bracket notation or the `cat` function. Treating the other object classes as subclasses of arrays has advantages. The code for the above common attributes can apply to several subclasses instead of recoding with possible discrepancies between classes. If the actual code cannot be reused for all subclasses, say, for reasons of efficiency, at least the concept along with its specifications can remain consistent. This gives system designers and programmers a gold standard to shoot for, namely, consistent behavior between classes. Once that is achieved, the user gains an advantage by having to learn one set of array conventions instead of many. In addition, the programmer can code with customized objects and operations that look like what they do, instead of being restricted to an unsuitable set of built-in conventions. In other words, object orientation encourages a more elegant style of program design, which in turn, eases both software maintenance and the users' learning task.

Because of the privacy conventions, good object-oriented programming style avoids global variables, which are a programmer's bane. Global variables are subject to inadvertent alteration by distantly-related parts of a program, creating bugs that are often hard to trace.

7.10 Other languages and comments

For further MATLAB information and examples, see the chapter "Classes and Objects" in "Using MATLAB". For another slant on object-oriented programming, see any book or manual on the language `C++`, e.g., Bjarne Stroustrup, "The C++ Programming Language", Addison-Wesley, Reading MA (1987), the original C++ bible. The language `Smalltalk` is preferred by some who consider themselves purists, as is CLU from MIT. (A Smalltalk overview can be found at the URL come.to/smalltalk, where various versions are rated.) The latest rage among languages is Java, from Sun Microsystems and IBM, among others. Recently (June 1998), on my monthly pilgrimage to White Flint Border's Books, I found a "wall" of shelves eighteen feet long by seven feet high, filled with object-oriented books and related matters. I will not attempt to evaluate such a literature, or even list it. The field is new, popular, and changing. Languages, systems, uses, and terminology are popping up, and sometimes dying, like flames in a forest fire. But most of the basics dealt with here are not going away. They will be built upon rather than replaced as the field matures.

7.11 A sample directory of object-oriented methods

In this section, we define a directory of the following Set methods, which will reside in the directory called `Set`:

```
1) Set.m       Define an object of class Set.
2) display.m   Permit display of contents of Sets.
3) double.m    Convert a Set to a row vector.
4) not.m       Define ~ for one object of class Set.
5) or.m        Define | for two objects of class Set.
6) and.m       Define & for two objects of class Set.
7) plus.m      Define + for two objects of class Set.
8) minus.m     Define - for two objects of class Set.
```

The goal of this directory is to provide algebra-like syntax for the following built-in set functions using the indicated infix operators:

```
union(|)  intersect(&)  setxor(+)  setdiff(-)
```

The class name `Set` is spelled with upper case S to avoid conflict with the built-in MATLAB handle-graphics function called `set`.

The global integer `SetMAX` must be initialized by the user before methods in @Set are used. `SetMAX` is the maximum integer permitted in Sets. Without `SetMAX`, the not ( ) operator and the Set-generating function `Set` will not work.

The functions (`double`, `not`, `or`, `and`, `plus`, `minus`) accept and produce single objects only, not arrays, and the function `Set` produces a single object. This does not prevent external use of Set arrays, as in `a(2,2)=Set(1:10);`, `b=[a;a];`, etc. That is, the inputs and outputs of these functions can readily become elements of Set object arrays. Because Set arrays are possible, the function `display.m` has been coded to accept arrays.

It is left as an exercise for the reader to redefine the following logical operators for Sets:

```
<   <=   >   >=   ==   ~=
```

to mean proper subset, subset, proper superset, superset, equivalent, and not equivalent, respectively. (Hint: use the built-in functions ismember and all.) For example, A¡B should produce logical 1 if all values in A are also in B, and B has at least one other value. Otherwise, A¡B should produce logical 0.

```
function y = Set( x );
% Set - Define objects of class 'Set'.
% A Set is a structure with a .set field containing
% a row vector of sorted positive integers with no repeats,
% Sets are subject to the usual operations of discrete set algebra.
% The operations defined on Sets are:
%    ~A    (not A)   : integers 1:SetMAX not in A.
%   A & B (A and B)  : integers in A that are also in B.
%   A + B (A xor B)  : integers in either set but not both.
%   A | B (A or B)   : integers in A or B or both.
%   A - B (A diff B) : integers in A that are not in B.
% NOTE: the global positive integer SetMAX is the upper limit
%       of the integer elements in Sets, which must be
%       initialized by the user.   If SetMAX is reduced, previously
%       defined Sets with elements > SetMAX may run into trouble.
% ---------- Input -----------------------------------------------------
% x = an object of class Set, or an empty array, or a
%     numeric array of positive integers, double or sparse.
% ---------- Output ----------------------------------------------------
% y.set = sorted x(:)' with repeats removed.
   % ---------- Begin -----------------------------------------------
   global SetMAX;                                % Global max integer
   if nargin == 0 | isempty( x );                % Should y be null?
      y.set = [];                                % Yes.
      y = class( y, 'Set' );                     % Convert y to a Set.
   elseif isa( x, 'Set' );                       % Is input a Set?
      if x.set(size(x.set,1)) > SetMAX;          % Check for legal max.
         error('Set element too large');         % Legal max exceeded.
      end;                                       %
      y = x;                                     % No conversion needed.
   else                                          % Input should be integers.
      if isa( x, 'sparse' );                     % Is x a sparse array?
         x = nonzero( x )';                      % Convert to double.
      elseif isa( x, 'double' );                 % Is x a full array?
         x = x(:)';                              % Make a row.
      else error('Non-numeric input to Set');    % Wrong class for x.
      end;                                       %
      x = unique( x );                           % Sort x, unique entries.
      nx = size( x, 2 );                         % How many x?
      if x(1)<1 | x(nx)>SetMAX | ...             % Is any x out of range
         any(logical(rem(x,1)));                 %   or non-integer?
         error(['Input to Set' , ...             % Error message.
            'must be +integers <= SetMAX']);     %
      end;                                       %
      y.set = x;                                 % Structure y gets x.
      y = class( y, 'Set' );                     % y is a Set.
   end;
```

```
function display(s);
% display - Display an array of objects of class 'Set'.
% This function is called whenever an object expression
% or statement is not terminated with a semicolon,
% resulting in display of the entire array in question.
% display can also be called directly, diplaying only the
% part of the array specified in the call, e.g.:
%       a(3) = b & c   % displays all of a.
%       display(a(3))  % displays only a(3)
   % ---------- Begin ----------------------------------
   nm = inputname(1);   % Get input name of Set.
   if isempty(nm),   nm = 'Output Set';   end;
   str1 = [ '   ', nm, '( ' ];
   str3 = ' ) =';
   d = size(s);         % Vector of dimensions.
   c = ones(size(d));   % Initial counters.
   e = c;               % Permanent row of ones.
   N = prod(d);         % # elements in s.
   for i = 1:N;         % Show each element.
      % --- Show Set name & subscripts ---
      if N == 1;   disp([ '   ', nm, ' =' ]);
      else   disp([ str1, int2str(c), str3 ]);
      end;
      si = prod( size( s(i).set ) );   % Size of contents.
      for k = 1:10:si;   % --- Show contents in rows of 10.
         if si > k+9;   v = s(i).set(k:k+9);
         else v = s(i).set(k:si);
         end;
         disp([ '      ', int2str(v) ]);
      end;
      % --- Update indices c ----------
      k = find( c<d );
      if isempty( k ),   break;
      else
         k = k(1);        % Index to be increased
         v = 1 : k-1;     % Indices to be reset.
         c(k) = c(k)+1;   % Increase current index.
         c(v) = e(v);     % Reset previous indices.
      end;
   end;
```

```
function y = double ( x );                % Appendix 2, page 4
% double - numeric conversion for objects of class Set.
% ---------- Input -----------------------------------
% x = object of class Set.
%     x.set contains a numeric row vector.
% ---------- Output ----------------------------------
% y = row vector, contents of x.set,
%     now suitable for ordinary numeric processing.
   % ---------- Begin ---------------------------------
   y = x.set;   % Convert Set to numeric row.


function y = not( x );
% not - Prefix 'not' (~) for objects of class 'Set'.
% For an object x of class Set, the expression ~x will produce
% the set containing all integers 1:SetMAX that are not in x.
% ---------- Input -----------------------------------------------
% x = object of class Set.
% SetMAX = global scalar integer, upper limit of the universal
%          set 1:SetMAX.  SetMAX must be initialized by the user.
% NOTE: If x contains elements that are not in 1:SetMAX,
%       an index-out-of-range error will occur.
% ---------- Output ----------------------------------------------
% y = object of class 'Set', containing all elements
%     of 1:SetMAX that are not in x.
   % ---------- Begin -----------------------------------------
   global SetMAX;                      % Use the global SetMAX.
   y.set = 1:SetMAX;                   % Initial structure y.
   y.set( x.set ) = [];                % Remove integers in x.
   y = class( y, 'Set' );              % Convert y to Set.


function z = or( x, y );
% or - Infix 'or' (|) for objects of class 'Set'.
% For objects of class Set, the expression x|y produces
% a Set with all integers in x or y or both,
% also called the set union of x and y.
% ---------- Input -----------------------------------------
% x, y = Sets.
% ---------- Output ----------------------------------------
% z = Set union of x or y.
   % ---------- Begin -----------------------------------
   z.set = union( x.set, y.set );   % Set union.
   z = class( z, 'Set' );           % Convert z to Set.
```

```
function z = and( x, y );
% and - Infix 'and' (&) for objects of class 'Set'.
% For objects of class Set, the expression x&y produces
% a Set with all integers in x that are also in y,
% also called the set intersection of x and y.
% ---------- Input ----------------------------------------
% x, y = Sets.
% ---------- Output ---------------------------------------
% z = set intersection of x and y.
   % ---------- Begin ----------------------------------------
   z.set = intersect( x.set, y.set );   % Set intersection.
   z = class( z, 'Set' );               % Convert z to Set.


function z = plus( x, y );
% plus - Infix 'xor' (+) for objects of class 'Set'.
% For objects of class Set, the expression x+y produces
% a Set with all integers in x or y but not both,
% also called the set exclusive-or of x and y.
% ---------- Input ----------------------------------------
% x, y = Sets.
% ---------- Output ---------------------------------------
% z = Set exclusive or of x and y.
   % ---------- Begin ----------------------------------------
   z.set = setxor( x.set, y.set );   % Set exclusive or.
   z = class( z, 'Set' );            % Convert z to Set.


function z = minus( x, y );
% minus - infix 'difference' (-) for objects of class 'Set'.
% For objects of class Set, the expression x-y produces
% a Set of all integers in x that are not in y,
% also called the set difference of x and y.
% ---------- Input ----------------------------------------
% x, y = Sets.
% ---------- Output ---------------------------------------
% z = set difference of x and y.
   % ---------- Begin ----------------------------------------
   z.set = setdiff( x.set, y.set );   % Set difference.
   z = class( z, 'Set' );             % Convert z to Set.
```

Chapter 8

Loops

Beware of loops. Where loops prove necessary, keep their content to a minimum. MATLAB is an interpreted language, and loops must be interpreted anew with every iteration. In contrast, expressions that merely imply loops (e.g. vector and matrix operations) may ultimately be executed by optimized code. In addition, many such expressions involve vector and matrix operations for which your machine may have special hardware to further speed things along. Such hardware will never be used if you insist on looping in an explicit way. The remainder of this chapter, and much of the next, is a case study in alternate looping strategies.

8.1 A bad example

```
Example 1:
function y = polyx(x,c)
% y,x are conformal matrices, c is a column vector.
% compute y(i,j) = sum_over_k(c(k)*x(i,j)^(k-1))
[rx,cx] = size(x);   rc = length(c);
y = zeros(rx,cx);
for i=1:rx,
   for j=1:cx,   y(i,j)=c(1);
      for k=2:rc,   y(i,j)=y(i,j)+c(k)*x(i,j)^(k-1);
         end;   end;   end;
```

The function `polyx` in example 1 evaluates a polynomial with coefficients `c(k)` for every element of `x`. The function `polyx` accepts matrix `x` and produces matrix `y`, so the caller can say:

```
y=polyx(x,c);
```

rather than:

```
for i=1:rx, for j=1:cx, y(i,j)=polyx(x(i,j),c); end; end;
```

Functions can accept and produce matrices, which saves the caller a lot of looping. However, this feature is not automatic. You must code your functions in matrix fashion to get any benefit. While the function `polyx` appears matrix-oriented to the user, it contains a triply nested loop (last 4 lines) that renders it exceedingly inefficient. We will reexpress this function with fewer or no loops.

8.2 Attacking the innermost loop

Let us start with the innermost loop on `k`. It multiplies each `c(k)` by `x(i,j)^(k-1)` and sums the terms. This is the inner product of the vector `c` with the vector of powers of `x(i,j)`:

```
Example 2: % replaces last 2 lines of example 1.
xp=x(i,j).^(1:k-1);   y(i,j)=xp*c;   end;   end;
```

But we are still generating `y` one element at a time.

8.3 A one-pass solution

Let us compute the vector `xp` in one pass for all `x`:

```
Example 3: % replaces the body of example 1.
rc=length(c);          % Polynomial degree + 1.
p=0:rc-1;              % Required exponents of x(i,j).
[rx,cx]=size(x);       % Dimensions of x.
x=x(:);                % Reshape x to a column vector.
x=x(:,ones(rc,1));     % Copies of x into rc columns.
p=p(length(x),:);      % Copies of p into length(x) rows.
xp=x.^p;               % Generate all powers of all x(i,j).
y=xp*c;                % Evaluate all the polynomials.
y=reshape(y,rx,cx);    % Reshape y conformal to input x.
```

This procedure is deliberately spread out so you can see the reason for each step. In practice, the last five lines of example 3 might be replaced by the single statement:

```
Example 4:
y=reshape((x(:,ones(rc,1)).^p(length(x),:))*c,rx,cx);
```

There are times when some degree of looping may be desirable in procedures like example 3. For example, if the dimensions of the above problem are `rc=rx=cx=100`, then the matrices `x`, `p`, and `xp` will require millions of bytes. Time will be used to form these matrices, and to allocate the storage space for them. When such matrices get large enough, MATLAB will spill them onto disk and retrieve them as needed, which requires lots more time.

8.4 Avoiding large matrices

Loops can avoid the formation of large matrices, as in example 2 and:

```
Example 5:
rc=length(c); [rx,cx]=size(x);
xp=ones(rx,cx);   y=zeros(rx,cx);
for k=1:rc,   y=y+c(k)*xp;   xp=xp.*x;   end;
```

In example 2, we loop on the dimensions of `x`, whereas in example 5, we loop on the length of `c`. A good way to choose between seemingly competitive codes is to run timing tests, as we do in the next chapter.

Chapter 9

Time and Work

9.1 Built-in measures

MATLAB provides some handy monitors of performance, two of which are `flops` and `tic-toc`. The function `flops` returns an estimate of the number of floating point operations used since the beginning of the session, or since the last time `flops` was reset by saying `flops(0)`. The function `tic` resets and starts the stopwatch, and the function `toc` returns the time in seconds since the last `tic`. The manner in which these features are used is best illustrated by example:

```
% Compare the computation effort in time and flops of
% statement #1: y=A*B*v, and statement #2: y=A*(B*v)
% where A & B are nXn matrices and v is an n-vector.
flops(0);                % Reset the flops count to 0.
tic;                     % Reset and start the stopwatch.
y=A*B*v;                 % Execute statement #1.
t1=toc;                  % Read the stopwatch.
f1=flops-18;             % Flops minus flops used by tic-toc.
% Use analogous code for statement #2. Print t1,f1,t2,f2.
```

9.2 Examples

Time and flops are by no means redundant measures. To illustrate, we will use the examples from the previous chapter, which are paraphrased in table 1 at the end of this chapter, with results in table 2. Note that the variation in flops in table 2 is within a factor of 2 for a given-sized problem, but the times differ by orders of magnitude. The times reflect not only raw computation, but also the overhead of scanning for-loops (very serious in polyx1), and of forming large matrices (the downfall of polyx4 despite the absence of loops). The focus in these examples is on RAM-based efficiency. Problem dimensions were kept within bounds so that intermediate matrices were not spilled onto disk. In general, such timings will depend on the platform being run.

9.3 Reliability of timings

Note that elapsed time may be an unreliable measure of your own computing time in time-shared (multi-task, multi-user) environments like workstations and mainframes. For those environments, MATLAB provides the function `cputime`, which returns your CPU time used since MATLAB started. However, on some platforms, even `cputime` may include uses of the CPU other than your own.

Here are some helpful suggestions about measuring running time. Use your multi-job computer at a time when no other jobs are running, say at 2am, when you are alone with the countless other people who are running time trials. Ignore the first run, whose timings may be contaminated by initialization issues like loading and compiling. Do several runs to account for time variation, if any.

9.4 Profiling M-files

It is often useful to determine which statements of an M-file are taking the most time. These statements might be a major inefficiency that can be corrected. Here is a code that will report the most time-consuming statements in a function called `func`:

```
profile func;
for i = 1:1000;
   y = func(x);
end;
profile report;
```

The loop is to ensure that sufficient time will be spent in the function to allow timing in seconds to be meaningful. Only one M-file can be profiled at a time. The report will list timings, and percentages of time, for a few of the most time-consuming statements in the M-file, totalled from all use of `func` between the two `profile` statements. For clarity, the times, percentages, statement numbers, and actual statements are listed across the page. The header of the report gives the total time spent in the function, and several statement numbers of the most costly statements in decreasing order. If you suspect that one of your M-files is a bottleneck of computing effort, the `profile` feature can help to find the cause. The frequency of use of the M-files themselves can be monitored with your own global counters incremented by each M-file on entry.

Table 1. Test function for tic-toc time and flops.
The function polyxt contains five subfunctions that do
the same calculation with different loop strategies.

```
function [ t, f ] = polyxt( x, c );
% Monitor time and flops performance of 5 polyx functions listed below.
% These functions compute a matrix of polynomial values y from a matrix
% of arguments x given the vector of coefficients c.
% Input  1: x = matrix of polynomial arguments.
% Input  2: c = column of coefficients of x^0, x^1, x^2, ...
% Output 1: t = row of tic-toc time for the 5 methods.
% Output 2: f = row of flops for the 5 methods.
   t = zeros(1,5);   f = t;   y = x;
   flops(0);  tic;  y(:,:)=polyx1(x,c);  t(1)=toc;  f(1)=flops-18;
   flops(0);  tic;  y(:,:)=polyx2(x,c);  t(2)=toc;  f(2)=flops-18;
   flops(0);  tic;  y(:,:)=polyx3(x,c);  t(3)=toc;  f(3)=flops-18;
   flops(0);  tic;  y(:,:)=polyx4(x,c);  t(4)=toc;  f(4)=flops-18;
   flops(0);  tic;  y(:,:)=polyx5(x,c);  t(5)=toc;  f(5)=flops-18;
function y=polyx1(x,c);   % ----- Triply-nested loop. ----------------
   [rx,cx]=size(x);   rc=length(c);   y=zeros(rx,cx);
   for i=1:rx;   for j=1:cx;   y(i,j)=c(1);
         for k=2:rc;   y(i,j)=y(i,j)+c(k)*x(i,j)^(k-1);
         end;   end;   end;
function y=polyx2(x,c);   % ----- Loop on rx and cx. ---------------
   [rx,cx]=size(x);   rc=length(c);   y=zeros(rx,cx);   p=0:rc-1;
   for i=1:rx;   for j=1:cx;   y(i,j)=(x(i,j).^p)*c;   end;   end;
function y=polyx3(x,c);   % ----- Loop on vector index. -------------
   [rx,cx]=size(x);   rc=length(c);   mx=rx*cx;
   x=x(:);   y=zeros(size(x));   p=0:rc-1;
   for i=1:mx;   y(i)=(x(i).^p)*c;   end;
   y=reshape(y,rx,cx);
function y=polyx4(x,c);   % ----- Big matrices but no loops --------
   [rx,cx]=size(x);   rc=length(c);   mx=rx*cx;   p=0:rc-1;
   x=x(:);   v1=ones(rc,1);   v2=ones(mx,1);
   y=reshape((x(:,v1).^p(v2,:))*c,rx,cx);
function y=polyx5(x,c);   % ----- Loop on rc only ------------------
   [rx,cx]=size(x);   rc=length(c);   xp=ones(rx,cx);
   y=repmat(c(1),rx,cx);
   for k=2:rc;   xp=xp.*x;   y=y+c(k)*xp;   end;
```

Table 2: Tic-toc time and flops for the five subfunctions in Table 1, using MATLAB 5.1 on a Sun Sparcstation 10 model 30. The inputs to polyxt are x=rand(n,n) and c=rand(n,1). Times are minima from five consecutive runs of polyxt.

n	<-------------- time in milliseconds ---------------->				
	polyx1	polyx2	polyx3	polyx4	polyx5
2	5	5	4	3	5
3	10	7	6	4	6
4	21	10	9	4	7
6	62	20	18	6	9
8	141	34	31	10	11
12	461	90	85	25	15
16	1077	172	164	58	20
24	3616	441	423	183	41
32	8556	893	869	459	80
48	28883	2484	2424	1577	217
64	68259	5305	5192	4030	470

n	<---------- flops ---------->				
	polyx1	polyx2	polyx3	polyx4	polyx5
2	16	25	26	26	13
3	72	82	83	83	55
4	192	193	194	194	145
6	720	649	650	650	541
8	1792	1537	1538	1538	1345
12	6336	5185	5186	5186	4753
16	15360	12289	12290	12290	11521
24	52992	41473	41474	41474	39745
32	126976	98305	98306	98306	95233
48	433152	331777	331778	331778	324865
64	1032192	786433	786434	786434	774145

Chapter 10

M-files

MATLAB code may be stored in M-files, i.e. files with a `.m` suffix. M-files come in two major categories: function files (or functions) and script files (or scripts). Functions are distinguished by an initial line beginning with the keyword `function`.

10.1 Workspaces

10.1.1 The master workspace

Your interactive master session has a workspace where your variables reside. If you call scripts from your master session (such calls can be nested, script calling script) those scripts share the master (or base) workspace. That is, if a script uses a variable `X`, and the script is called from the master session, it will use `X` from the master workspace, provided `X` is there. If the called script creates `X`, it will be placed in the master workspace, where you have access to it.

10.1.2 The global workspace

In addition to the master workspace, your master session may have access to the global workspace, which contains all variables that have been declared global somewhere, either in the master session, a script, or a function. Once your master session or any of its called scripts issues a `global X` command, you will be using `X` from the global workspace. In addition, any functions that contain a `global X` command will also be using that same `X`. The purpose of the global workspace is to enable the master session, scripts, and functions to share variables without passing those variables as arguments. Here are some common uses for global names:

- Names of M-files and built-in functions are global (in the sense that they are available anywhere), except for M-files that reside in private directories. Names of other files, e.g., data files, in the MATLAB search path are also global.

- To profile the frequency of M-file use, you can set up global counters named `NRUN` followed by the underline symbol (`_`) and the function name, e.g. `NRUN_func`. The master session would initialize and monitor the values of such counters. The first statements in any such M-file would be the global statement followed by the counter update:

```
function y = func(x);
global NRUN_func (other global stuff goes here);
NRUN_func = NRUN_func + 1;   % Increment counter.
```

- If a huge array must be updated frequently, with only a few entries changing at a time, it is time-consuming to submit the array as an argument to the function that will do the changes. Because the function is changing the array, however slightly, MATLAB must find space to present a complete (and massive) copy of the array to the function. Placing the array in the global workspace avoids the problem.

- If you are running a lengthy iterative process that sometimes aborts or goes on forever, you can place the best results so far in global variables, where you can recover them when the process aborts or when you interrupt it.

Unnecessary use of the global workspace is ill-advised because of the danger of unintended duplicate names. To reduce the probability of such mishaps, global names should tend to be long (at least four letters) and should feature LOTS of upper case letters, in contrast to your other variable names.

10.1.3 Function workspaces

In addition to master and global workspaces, each function has its own function workspace in which its nonglobal variables reside. This workspace is created when the function is called, and disappears when the function returns to the caller. The same pattern is observed when functions call other functions, including subfunctions in the same file. Functions do not share variables of the same name with each other or with the master session unless those variables are global. All nonglobal communication of variables between functions is in the form of input and output arguments. However, functions can call scripts, in which case the scripts use the calling function's workspace rather than the master workspace. Unlike functions, scripts always share the workspace of the calling process, which is their basic way of receiving and passing on variables in lieu of arguments.

10.1.4 Accessing other workspaces

Deeply nested function calls create a series of workspaces, but only the workspace of the current (most recently called) function is active. The others are dormant. Two of these dormat workspaces are of special importance: the master (or base) workspace, and the workspace of the process that called the current function, i.e. the caller workspace. On occasion, we may want an invoked function to have access to these other workspaces. The built-in function `evalin` allows functions to reach into the caller or base workspaces:

```
evalin( 'caller', 'x=6;' );   % Change caller's x.
evalin(  'base',  'y=7;' );   % Change base's y.
```

The built-in function `inputname` allows functions to get the caller's names of the input arguments. This enables a function to inform the caller about certain variables using the caller's terminology. Unfortunately, these two facilities can negate the usual precept that, except for the outputs, a function should not disturb the caller's workspace. The function `evalin` coupled with `inputname`, just like the `global` feature, should be used with considerable caution.

10.2 Self-documentation

When commenting M-files, it pays to follow protocol. In scripts, the first lines should be comments with `%` as the very first character of each line (no leading blanks, no intervening blank lines, no intervening statements). In functions, these lines must immediately follow the function statement. These comments constitute the help file for this M-file, so that typing `help F` will display the initial comments of the M-file `F`, ending at the first line that does not have `%` as its first character. The first comment line, called the H1 line, should consist of the M-file name followed by a terse description of what the M-file does. If you enter a directory using the MATLAB `cd` command, and type `help dir`, where `dir` is the directory name, all H1 lines in that directory will be displayed. This is one good reason to group related M-files in separate directories. Alternately, you can put a `Contents.m` file in the directory, which will be displayed instead of the H1 lines. Also, the `lookfor` command scans H1 lines.

The initial comments should always contain the following:

1. An initial H1 line.
2. An adequate explanation of what the M-file does.
3. A list of input arguments and their meanings, indicating whether they are essential or optional, and what their default values are. If some input arguments can be omitted, tell what happens if they are. For scripts, these comments should list any variables that have to be predefined in order for the script to work.
4. A list of output arguments, as above. For scripts, these comments should list any variables that are created or altered by the script. Scripts have the nasty habit of leaving the variables they create in the workspace. When coding scripts, make liberal use of the `clear` command to avoid a crowded list of variables, and to avoid having to explain their trivial nature in the initial comments.
5. A list of additional functions and scripts required by the M-file.
6. An explanation of *how* the M-file does its job, especially if the methods are good for some cases and not for others.

Beyond the initial comments, a step-by-step commentary is often essential for others to understand your code. Even you will be helped by such comments, when looking back on your own code that hasn't been seen for a while. My rule of thumb is: the right side of the page is reserved for comments, usually one comment per line of code. Code stays on the left, using ellipses and continuations where needed to keep code from impinging on comment space. Occasional detailed comments take entire lines. More comment than code is a good thing.

10.3 Function arguments

The first line of a function, called the function line, is of the form:

```
function [ FOA ] = funcname( FIA )
```

where FIA (function input arguments) and FOA (function output arguments) can be absent or single arguments or a list of arguments separated by commas. If FIA is absent, omit the parentheses. If FOA is absent or a single argument, omit the square brackets. If FOA is absent, omit the `=` symbol too.

A function may be invoked by a separate calling statement:

```
[ COA ] = funcname( CIA )
```

where CIA (caller input arguments) and COA (caller output arguments) have the same syntax as FIA and FOA. CIA and FIA do not have to match in name or number. Order alone tells which caller input corresponds to which function input, and similarly for COA and FOA. A function that returns at least one argument can also be invoked within a caller's expression, as in `y=2*F(x)+5`. In this context, the caller is understood to be asking for a single output of some sort from function F: array, structure, or whatever.

A function communicates mainly through its arguments. Some functions allow an arbitrarily large number of arguments. Other functions allow the caller to omit some arguments. The function's documentation should make clear when such actions are permitted, and how the function interprets them. The actual number of arguments the caller used is given by the function `nargin` for input, and `nargout` for output, which can be used by the function to react appropriately when the number of arguments varies.

It is MATLAB's usual policy that a caller's variables should not be subject to change when they are included in an input argument list. If an input argument is not altered by the function, only the argument's address is passed to the function to avoid making needless copies of arguments. In this case, the function uses the original data. On the other hand, if an input argument is altered in the function or the scripts that it invokes, a copy of the argument is used, so that the caller's copy remains unaffected. For exceptions, see the section titled "Workspaces" above.

10.4 Variable numbers of input arguments

Sometimes, it is impossible to anticipate how many arguments a caller will submit to a function. As a case in point, consider a root-finder function (call it `Root`) that accepts the name of second function (call it `F`) and a scalar `X`, the object being to find that value of `X` that will yield `F(X)=0`. `Root` will call `F` repeatedly, varying `X` until `F(X)` is zero or very small. The problem is that, often, functions like `F` need other values, parameters if you will, as well as the principal variable `X`, and the number of such parameters depends on the nature of the problem. Rather than put these parameters in the global workspace, we would prefer to pass them as arguments to `Root`, which in turn could pass them to `F`, without `Root` caring about how many such parameters there are.

The task is accomplished using the argument `varargin`. Let us say that `F` requires three arguments, the variable `X`, plus the two parameters `Y`, and `Z`, so that `F(X,Y,Z)` is the required call. The call to `Root` might be:

```
Xfinal = Root( 'F', X, A, B, Y, Z );
```

indicating that `F` is the function to be zeroed, with respect to `X` between limits `A` and `B`. The first four arguments are really all that `Root` cares about. Internally, here is how `RF` treats its arguments:

```
function Xroot = Root( Fstr, X, A, B, varargin );
% The while-loop that varies X begins here.
% The following statement calls function whose
%  name is in Fstr with first argument X and the
%  remaining arguments from elements of varargin:
   FX = feval( Fstr, X, varargin{:} );
% The remainder of the while-loop goes here.
```

In the first line, `varargin` is a cell array, into whose elements have been packed all the extra arguments that the caller may have provided. In the `feval` line, the expression `varargin{:}` is the list of *contents* of the cell array, i.e., the last part of the caller's input argument list to `Root`, which now becomes the last part of `Root`'s input argument list to `F`. Note that the extra arguments `Y` and `Z` from the caller are never explicit in `Root`'s internal code. Finally, the function line of `F` can be coded as follows:

```
function fx = F( X, Y, Z );
```

oblivious to the fact that `varargin` is acting as a go-between.

If the extra arguments are to be used directly, rather than passed to another function, there are several ways to handle the situation, First, the function statement can include many arguments:

```
function y = F( Arg1, Arg2, Arg3, ...
                Arg4, Arg5, Arg6, ...
                   and so on );
nArgs = nargin;
```

The argument list is continued for as many arguments as the function `F` is ever expected to receive in a single call. In this context, the calling statement can provide fewer arguments than the function statement allows, but not more. The function `nargin` tells `F` how many arguments the caller provided this time, so that `F` can be coded to handle any possibility, except too many caller arguments. Any `Arg`'s that the caller does not provide will be undefined in the function `F`.

Another method for varying numbers of arguments is to use `varargin` as a final argument, to package the end of the caller's argument list in a cell array, as in the function `Root` above. This device places no upper limit on the number of caller arguments. However, if `Root` is to use those arguments directly, it must somehow unpack the cell array, e.g.:

```
Arg1 = varargin{1};   Arg2 = varargin{2};   etc.
```

This device has a danger. It allows the caller to provide more inputs than the function is coded to accept, possibly concealing a bug. The extra elements of `varargin` will simply go unused, with no error message to that effect. For this reason, functions that use `varargin` should check the `nargin` function to see how many inputs there really are.

Another alternative is for the caller to package his arguments in cell arrays or structures, to be unpacked by the called function. One advantage of this approach is that arguments often come in several separate categories. Each category of argument can be packed into a distinct cell array. A possible disadvantage of such packing is that it creates copies of possibly voluminous input data. Unlike varargin, which must be the last item in the function line, prepackaged cell and structure arrays can be entered in any agreeable order, as long as that order corresponds between the call and function statements.

10.5 Variable numbers of output arguments

While the caller can ignore function output arguments by omitting them from the output list in the calling statement, he cannot ask for more outputs than the function provides. However, the function can be coded to provide an arbitrarily large number of outputs using the `varargout` feature (essentially the reverse of the `varargin` feature described above). The cell array `varargout`, if used, must be the final function output argument, and it must be defined within the function, e.g.:

```
function[varargout] = Hcube(x);
% Hcube - produce matrices of hypercube vertices
% of dimensions 1:nargin and sides of length x.
cube = [ 0; x ];                % Put vertices of 1-D
varargout{1} = cube;            %  cube in 1st cell.
v0 = 0;   vx = x;               % Init. new column.
for j = 2:nargout;              % For jth cell,
   v0 = [v0;v0]; vx=[vx;vx];    %  set up new column,
   cube = [ cube, v0; ...       %  up previous dimension
            cube, vx ];         %  of cube by 1.
   varargout{j} = cube;   end; % Pack output.
```

If the caller wants the coordinates of, say, all unit hypercubes up to the fifth dimension, the calling statement should read:

```
[c1,c2,c3,c4,c5] = Hcube(1);
```

This example is designed to be simple enough to work easily by hand, so you can see what the code does. In the function line, `varargout` does not require the index `{:}`. MATLAB distributes the contents into the outputs anyway, one cell of `varargin` per output. In the fifth line, the function `nargout` tells how many outputs the caller has specified, just as `nargin` does with inputs. In this application, the jth output contains $(j+1)2^j$ numbers, so if the caller asks for, say, 20 outputs, there just might be a lack of workspace.

10.6 The keyboard feature

The `keyboard` statement stops execution within a function, allowing the session owner to examine and change values that are ordinarily private to that function. In fact, while the `keyboard` statement has control, the session owner can issue any MATLAB statements that are legal at that point. This is a powerful debugging tool, and it can also be used as an input device. When a `keyboard` statement is encountered, it retains control of the session until the user issues a `return` (spelled out). At that time, the function resumes execution from the stopping point. (Authors bias: We would prefer a blank response instead of `return`, especially for keyboard statements that are iterated frequently.) The documentation says "all MATLAB commands are valid" when a `keyboard` statement has control. However, some commands may not do what you expect. For example, when the keyboard statement is in a for-loop:

```
for j=lo:del:hi,   keyboard;   end
```

Changing the values of `lo`, `del`, or `hi` with this `keyboard` statement will not change the sequence of values assumed by `j`. The looping vector `lo:del:hi` is preset before the loop executes. Also, editing the function from within itself, i.e. editing the `.m` file on disk from a keyboard statement within the function, will not affect the current call:

```
function y=multpl(x)
keyboard; y=[x,x];
```

If, from the above `keyboard` statement, you edit the file `multpl.m` so that the last statement reads `y=[x,x,x]`, `y` will still get two copies of `x`, not three. `y` will get three copies of `x` on the next invocation of multpl. Allowing a user to alter the state of a program in mid-run is a tricky business at best. But even with these occasional surprises, the keyboard feature is a gem.

10.7 debugging

Using `keyboard` statements for debugging is handy when you know where to place them. But too often, bugs arise where they are least expected. For this contingency, MATLAB provides `db` statements, `db` being short for `debug`. The statement `dbstop if error`, issued in your master session, is especially useful. When an error occurs in a function, this statement puts you in `keyboard` mode, near the point in the code where the error arose. The effect is the same as if you had placed a `keyboard` statement in that same spot, with exactly the same priviledges.

10.8 Error messages

Certain errors will terminate the execution of statements, scripts, or functions, and error messages will be issued. These things will happen with no explicit request on your part. In this user's experience, MATLAB error messages are, on average, far more informative than messages from compiled languages like FORTRAN. Usually, the very statement in which the error occurred is identified, along with the type of error like `zerodivide`. You should appreciate that such service has a cost. Somewhere, pieces of code must be tracking, and counting, and checking. Design decisions must be made to insure that such monitoring activity is efficient, and that it doesn't become a prominent part of your computation cost.

Your appreciation for such activity will grow as you write your own files. Because then, you will have the option to create your own error messages in addition to those from MATLAB. You will have to decide for yourself how much error-checking to do. Remember: the scariest part of a project is when the error messages stop coming. Be as thorough in your MATLAB self-criticism as it is practical to be. Any less may save you seconds initially, only to cost you weeks when something goes wrong.

As an example, consider a function `f(A,B,C)` of three real square matrices of equal size. You have decided to check the following:

Are there exactly 3 arguments?
Are they non-empty?
Are they square arrays?
Are they of equal size?

You have decided not to check for complex or non-numerical arrays, because these will cause other error messages as the computation proceeds.

```
function D = f( A, B, C )
if nargin~=3, error('f expects 3 arguments'); end;
if isempty(A), error('empty argument'); end;
vs = [ size(A), size(B), size(C) ];
if length(vs) ~= 6  |  any( vs(1:5) ~= vs(6) ),
   error('f expects square matrices of equal size');
end;
```

Note how all of the desired checks will be done. Although the error message may be slightly less than specific, it will be specific enough for the caller to spot the problem. You could skip, say, the check on equal size, in the conviction that non-conformal matrices will cause errors further into the code. But if the resulting message is likely to confuse the user (especially if the user isn't familiar with the function code), then catching the error at the outset with a user-friendly message is worth while.

If a frequently iterated function has time-consuming error checks, you may be wasting time on largely redundant checking. By the same token, if the checks are removed entirely, you may be wasting time computing nonsense. The question is not whether to check, but where and when. It may pay to move some of the error checks from the function itself to a less busy spot in the code. For example, if the function `g` is being invoked from within nested loops of another function `f`, some of the error checks for `g` might well be placed outside the innermost loops of `f`.

10.9 Warning messages

In certain cases, MATLAB elects to issue warning messages, which, unlike error messages, do not stop execution. They simply make you aware of suspicious conditions. You can also issue warning messages:

```
if datestr(D,8)=='Mon',
   warning('User beware! My code makes mistakes on Mondays.');
end;
```

Chapter 11

Tricky stuff

Any language as rich as MATLAB is bound to have its confusing aspects, e.g., code that looks like it does one thing when in fact it does something else. A review of some of these will help you to solidify your mastery of MATLAB and to avoid bugs.

11.1 Indexed Targets

The following effects are discussed with numerical arrays in mind, but most of these effects also occur with non-numerical arrays. Let `T` be a target, and let `J1,J2,J3,...` be ordinal indices. A statement of the form:

```
T(J1) = X  or  T(J1,J2) = X  or  T(J1,J2,J3,...) = X
```

may have one or more of the following effects:

1. It may assign values to existing elements of `T`.
2. It may convert numerical `T` from real to complex.
3. It may create new `T` of sufficient size.
4. It may expand pre-existing `T` to sufficient size.
5. It may assign new zero or empty elements to `T` by default.
6. It may delete elements of `T` if `X` is null.
7. It may reshape `X` to fit in `T`.
8. It may regard T as an array of fewer dimensions.

What a wondrous entity an indexed target is, that it can accomplish so much. But with every added capability comes added danger. Let us retrace the above list of effects with corresponding errors, any of which may slip by without detection at the time of occurence, and in later calculations as well:

1. Storing an entry with an incorrect value. For numerical arrays, MATLAB will accept any values including NaN, Inf, and complex numbers.

2. Storing an erroneously complex number, e.g. from `T(i)=sqrt(x)` where `x` should have been positive. One complex entry causes the entire array `T` to be stored as complex.

3. Storing `X` in nonexistent `Tl` (tee-ell) when it should have been stored in pre-existing `T1` (tee-one). MATLAB creates `Tl`, and stores `X` there.

4. Addressing `T` with an erroneously large index entry. MATLAB expands the dimensions of `T` to fit.

5. In the same error as above, creating spurious zero entries in the expanded `T`.

6. Storing an erroneously empty matrix `X`. When `X` is stored in a non-empty subvector of `T`, that subvector disappears from `T`.

7. Creating a 2-by-2 matrix `X` when it should have been a row 4-vector (a single unintended semicolon can do this) with the intended vector in row-major order. When `T(1:4)=X` is executed, `X` is reshaped to a 4-vector, but now the elements are in the wrong (column-major) order.

8. Accidentally using a single subscript for array `T`. MATLAB regards `T` as a vector. As a result, the new entries are accepted, but placed in the wrong elements.

Clearly, proper target indexing calls for some caution. There are many ways to err that will not elicit an error message, but will simply proceed erroneously.

11.2 The diag function

Given an m-by-n matrix `X`, where `m>1` and `n>1`, `diag(X)` produces the column vector of main diagonal elements of `X`. Inversely, if `X` is a vector, `diag(X)` produces a diagonal square matrix with `X` on the main diagonal. A conflict arises when the shape of a matrix argument can vary, i.e. when we want to regard `X` as a matrix and extract its diagonal, even when `X` is a vector. To avoid forming an unwanted matrix, a test is needed:

```
if min( size(x) ) == 1,   d = x(1);
else                      d = diag(x);   end
```

The function `size(X)`, when used in this manner, produces a 2-vector containing the dimensions of `x`.

The conflict can also be avoided by using two functions: `diagin`, which always inserts a diagonal in a matrix, and `diagout`, which always extracts a diagonal vector. The user can code these easily, and thereafter not deal with this problem:

```
function M=diagin(v); % Create matrix M with diagonal v.
if min(size(v))>1, error('vector argument required.'); end;
M=diag(v);

function v=diagout(M); % Extract diagonal v from matrix M.
if min(size(M))==1, v=M(1); else v=diag(M); end;
```

In MATLAB 5, `diag` is more versatile, having an optional second argument. The coding of `diagin` and `diagout` with two arguments in MATLAB 5 is left as an exercise for the reader.

11.3 The sum function et al.

Several MATLAB functions, e.g.:

```
any all max min sum prod mean median std cumsum cumprod diff
```

work on vector input. In a quite natural way, these functions have been extended to operate on arrays, by operating on the first non-singleton dimension, thus producing an another array (possibly a scalar or vector) of results. However, the output can have surprising shape. The following example illustrates, where j is a scalar:

```
R = rand(j,7);   S = sum(R);
```

`R` is a `j`-by-7 matrix, and `S` is a row 7-vector of the column sums of `R`, unless `j=1`, in which case `S` is a scalar sum of the elements of the row vector `R`. To avoid such surprises, a multi-argument form of the above functions, provided by MATLAB, should be favored. The first argument to `sum` may be an array of arbitrary size and dimensionality. Typically, the second argument is the dimension over which you want the function to operate. The output of `sum(A,N)` is an array of at most the same dimensionality as A, but with the dimension indicated by N reduced to 1, e.g.:

```
R = rand(I,J,K);    % R is a 3-dimensional array.
S = sum(R,2);       % S is dimensioned (I,1,K)
T = rand(I,J);      % T is a matrix.
U = sum(T,1);       % U(j) = sum(T(:,j)) even if I=1.
```

This convention has not been extended to sums, etc., over more than one dimension at a time. However, to sum an entire array `A`, use `sum(A(:))`.

The functions listed above do not behave identically with respect to their arguments and the size of their output. In particular, consult the help file on:

```
cumsum cumprod diff max min
```

11.4 Vectors defined by colon notation

For positive `h`, the statement `t = t1 : h : tn;` defines a vector with the `n` successive values `t1`, `t1+h`, `t1+2*h`, out to some final value `t1+(n-1)*h` where `t1+(n-1)*h <= tn` and `t1+n*h > tn` are both true. The final value `t1+(n-1)*h` must lie in the closed interval [t1,tn]. For example, `6:2:9` means the row vector `[6,8]`. So far, we're talking about the literature definition. However, the designers are aware (as we all should be) that the arguments `t1`, `h`, and `tn` are often subject to roundoff error, as are the intermediate elements of the vector. So they have made some compromises that violate the strict interpretation of the rules. When the final value of the vector comes very close to but just beyond `tn` (relative to the range of the vector), it is regarded as exactly `tn`, which becomes the final value. If this were not done, some seemingly simple constructs would produce unexpected results. For example, the vector `7/25 : 7/25 : 7` should have 25 elements, but in IEEE floating point arithmetic, `7/25` is slightly too large, so that under the strict cutoff rule, the vector would only get 24 elements. The cutoff rule is relaxed to avoid such unintended truncations. A second kind of rule-bending sometimes occurs when an element of `t` lies very close to zero. Sometimes it will be "rounded" to zero, and sometimes not.

In cases where the arguments have sizable numerical errors, the `linspace` function may be preferable to colon notation, because with `linspace`, you always get exactly the number of elements you expect at exactly the endpoints you specify.

11.5 Unenclosed arguments

If the arguments to a function are character strings, and there is no assignment involved (i.e., the function stands alone), then there are two ways to invoke the function. For example, consider the `clear` function:

```
clear('x')   % Enclosed explicit character string.
clear x      % Unenclosed implied character string.
```

Either statement will remove `x` from the workspace. However, the second form of statement will work with functions that really shouldn't accept string arguments:

```
log10(d)     % Proper way to show log10 of variable d.
%------------------------------------------------------------
log10 d      % Here, d is a character with internal code 100,
log10('d')   % making these last 3 versions equivalent.
log10(100)   % The result will always be 2, regardless
             % of the status of any variable named d.
```

In publications and worksheets, functions are often written with unenclosed arguments. Avoid that convention in MATLAB. Always enclose the argument when asking for a function value.

11.6 Appending and removing vector elements

In an application where parts of a column vector are being appended and removed in unpredictable fashion (e.g. queuing models), some caution must be taken in those cases when the vector can become scalar or empty:

```
q = [1;2;3;4];  % At opening, 4 customers are waiting.
c=4; n=4;       % c=# customers so far, n=length of queue.
q(1:3) = []; n=n-3;    % 3 customers are served before
n=n+1; c=c+1; q(n)=c;  %    customer #5 shows up.
```

Line 3 of the above code shortens `q` to a scalar. Line 4 makes `q` a vector again, but this time `q`, having no memory of its previous column status, becomes a row. The obvious fix is to code `q` as a row at the start, or if the column status is preferable, change `q(n)` to `q(n,1)` in line 4. A similar problem occurs when `q` becomes empty, i.e., all customers served before the next ones show up.

11.7 Appending and removing fields

A different problem arises in a structure or structure array, when fields are appended and removed:

```
% x = structure with fields f1, f2, ..., fn.
% n = number of fields in x.
x = rmfield(x,['f',int2str(n)]);   n=n-1;   % Remove field fn.
n=n+1;   eval(['x.f',int2str(n),'=[]']);    % Append field fn.
```

If `x` is a structure with several fields, line 3 removes a field from `x`, and line 4 appends a field to `x` But when `x` starts with only one field, line 3 produces an empty matrix, not a structure. Line 4 becomes illegal, because you can't append a field to an empty matrix.

If you insist on the "append and remove" approach for fields, test for empty "structures" and code around them:

```
if isempty(q);           % If q has no fields,
   q=struct('f1',[]);    %   create a new structure and
   n=1;                  %   start the field count.
else                     % Otherwise,
   n=n+1;                %   bump the field count and
   eval(['q.f',int2str(n),'=[]']);   % append field fn.
end;
```

11.8 Variables *versus* .m files

An expression of the form `f`, or `f(i)`, or `f(i,j)`, etc., might be an indexed variable or a function call. If, say. you put such an expression in a script file, assuming that the supposed variable would be initialized elsewhere, you may be invoking a function accidently. Note that a function call does not need an argument list, so even the use of what appears to be a simple scalar like `xyz` may invoke a file of the same name if the scalar `xyz` has not been initialized. Another error of this type occurs if you use the name of a built-in value like `eps`, thinking that it is your variable, but forgetting to initialize it. A similar error occurs when you clear such a variable, then (erroneously) try to use it. For example, the built-in value of `pi` is always there, even after you've cleared your variable called `pi`.

Even script files are not immune to this confusion. One way to print the value of a variable is to type its name alone on a line with no semicolon. However, if you forget to define that variable, and there is a script file with the same name as the supposed variable, odd things can happen. For example, you can get an expected yet spurious printout, along with some unwanted computation, causing several of your variables to change value mysteriously.

11.9 The eval and feval functions

`eval(STR)` treats the string `STR` as MATLAB code, and executes it. This feature appears useful for receiving strings as arguments to your functions, and using them as function names, but eval is not safe in this context:

```
% Generate a function of A, plus random error.
function M = g(funcname,x)
% funcname is a string, x is scalar.
% Compute Lotsa_stuff here.
M = eval([funcname,'(x)']) + Lotsa_stuff;
```

The string `funcname` above is intended to contain the name of a function to be evaluated with argument `x`. But if `funcname` contains the string `'x'` or any other name internal to the file `g`, then funcname will be misinterpreted as a variable name. Thus it might produce a diagnostic (in good cases), or a wrong result that might be hard to detect. This becomes more probable if the function `g` is used as a black box by many people. MATLAB provides a safer construct for such uses: `feval`, which accepts `funcname` and an argument list, and which forces `funcname` to be interpreted as a function, not as a variable. So, to fix the above example, change the last line to:

```
M = feval(funcname,x) + Lotsa_stuff;
```

11.10 The global declaration

Ordinarily, a variable `x` used within a function does not interact with a variable `x` used in the master session or in some other function. The statement `global x` can alter this convention, making it possible for functions and the master session to share the same variable. Any use or alteration of global `x`, even within functions, refers to that single global value. Clearly, this is dangerous, because functions you didn't write but which you are using as black boxes, may also be using a global version of `x`.

The `global` feature should be used sparingly to avoid unwanted interactions. Functions intended for use by others as black boxes should LOUDLY document any global variables they use. See the chapter on side-effects for further comments about `global`.

When a variable `x` is global, there are two ways to "clear" it. The statement:

```
\verb#clear global x#
```

removes `x` entirely. However, if another function is then invoked that contains the statement `global x`, `x` will be reinitialized as an empty matrix. Where there is one `global` statement, there are usually others. (The statement `global x` serves no purpose unless `x` is also declared global somewhere else.) So it is not safe to rely on the non-existence of a global quantity, even if you have cleared it.

In contrast to `clear global x`, the statement `clear x` removes `x` only from the local workspace. Its current value remains in the global workspace, and while it is now unaccessible to the local code, it remains accessible to other functions that declare `x` global. That is, the `clear x` statement causes the local code to forget `x`, and its global status as well. The local code can now create a new local variable `x`, with no effect on the global `x` value.

```
% Example:
global x;           % Global x exists, but is empty.
x = [ 1,2; 3,4 ];   % Global x is 2-by-2
                    % Global & local x are the same.
clear x;            % Local x is now undefined.
                    % Global x is still 2-by-2.
x=5;                % Local x is scalar, global x is 2-by-2.
global x;           % Local code will once again
                    %  use global x (2-by-2).
```

In MATLAB 5, the local code can reconnect to the global value of `x` by issuing another `global x` statement, as in the last statement above. This reconnection may be impossible in future versions of MATLAB, so avoid it. In general, it is best to keep local and global variables distinct. Choose names for global variables that you will never use for local variables, e.g. begin all global names with `GLOB`, as in `GLOBxyz`.

11.11 Empty *versus* nonexistent variables

An empty variable may be created e.g. by statements like `X = []`. Empty variables exist, they have the defined value "empty", and they may be used in expressions and calling sequences. Nonexistent variables may be "uncreated" by never initializing them, or by clearing them, e.g.: `clear X Y`. One caution about uninitialized values: some "scalars" are really functions without arguments, provided by MATLAB, e.g., `i, j, pi, eps`, so you can never rely on the nonexistent status of such "scalars", even if you clear them. Similar caution is recommended for global variables, as described in the previous section. In contrast to empty variables, nonexistent variables are not legal symbols in expressions. However, this distinction has not been strictly adhered to in MATLAB 5, so be careful. Here are some examples:

```
clear x;   x = sin(x);   % Result: error message.   Good!
clear x;                 % Same statements on separate lines.
x = sin(x);              % Result: x = [];   Bad!
clear x;
x = [ 1, 2, x ];         % Result: x = [ 1, 2 ];   Bad!
```

True, MATLAB 5 does produce warning messages in these cases, but in a run full of output, these can go unnoticed. (Warnings, after all, don't stop execution, because they often don't indicate a bad result.) The bottom line is that legitimate-looking results can be produced from undefined quantities. Presumably, future versions of MATLAB will eliminate these anomalies.

11.12 Long variable names

A variable name can be very long, but MATLAB will recognize only the first 31 characters. Two variables with very long names which differ only beyond the 31st character will be regarded as the same variable:

```
% A 'proof by MATLAB' that 1 + 2 = 4:
Variable_with_a_Very_Long_Name_1 = 1;
Variable_with_a_Very_Long_Name_2 = 2;
x = Variable_with_a_Very_Long_Name_1 ...
  + Variable_with_a_Very_Long_Name_2;      % Result: x = 4.
```

11.13 Delimiters in matrix definitions

Blanks may be used rather freely within expressions, unless those expressions are within square brackets, as definitions of matrix elements or submatrices. In the latter context, a blank may or may not be an expression delimiter, possibly creating more elements than you might expect. Thus we have the following collection of results:

```
              Row vector                Elements

[    x*y+u*v,      x*y-u*v  ]       2
[   x*y +u*v,     x*y -u*v  ]       4
[  (x*y +u*v),  (x*y -u*v) ]       2
[   x*y + u*v,    x*y - u*v ]       2
```

In such explicit matrix definitions, MATLAB 5 uses the spaces around an operand to decide whether that operand is a unary sign as in `x +y` (producing two separate elements `x` and `y`), or a binary operator as in `x+y` or `x + y` (producing a single sum). If you want complete freedom to use spaces in your elements, enclose those elements in parentheses.

The end-of-line also behaves in some odd ways. For example, consider the two matrix definitions (listed side by side to save space):

```
G = [ [ x1, y1 ];          H = [ [ u1; v1 ],
      [ x2, y2 ];                [ u2; v2 ],
      [ x3, y3 ] ];              [ u3; v3 ] ];
```

The user's intention is to produce a 3-by-2 matrix `G`, using semicolons to stack the rows (which works), and a 2-by-3 matrix `H`, using the commas to place the three column vectors side by side (which fails). Were `H` entered on one line, or were the first two lines ended with ellipses, `H` would indeed be 2-by-3. Alas, neither is done, yet MATLAB 5 accepts the definition of `H` without complaint. However, ends-of-line override the commas in `H`, producing a column 6-vector instead of a 2-by-3 matrix.

11.14 Dots

Dots (periods) are used in the following manner:

1 dot for decimal point,
1 dot for array operations (`.*  ./  .^`),
1 dot for non-conjugate transpose (`.'`),
1 dot for structure field separation, e.g. `f1.f2.f3`
2 dots for parent directory,
3 dots for ellipsis (line continuation).

Despite the numerous uses for dots, conflicts of meaning are actually hard to come by. MATLAB 5 prefers to interpret a dot as other than a decimal point if it can, so that `2.^X` indicates an array of powers, not `2.0` raised to the matrix `X` power. Still, if there is a possible conflict of meaning, use spaces for clarity. For example, let `X` be a matrix:

```
A = 2. ^ X;       % A gets the matrix exponential 2^X.
B = 2.^X          % Each element B(i,j)=2^X(i,j), unclear.
C = 2  .^  X;     % Each element C(i,j)=2^X(i,j), clear.
```

The ellipsis is a peculiar beast. Usually, when coding a multi-line statement, you will get into trouble if you don't use it. But occasionally, you will get into trouble if you do, particularly when assigning sizable matrices explicitly. Rows of a matrix that occupy more than one line of code must be continued with an ellipsis. But even if you end all non-final matrix rows with semicolons, you should begin each new row of the matrix on a new line, *without* an intervening ellipsis. With ellipses, MATLAB may complain about the size of your statement, yet without ellipses, the same statement is processed without fuss. This oddity is encountered when the defining statement is large, where the definition would most likely be stored in a `.m` file.

11.15 The "end" statement

In several languages, `end` statements terminate the scope of certain control statements. In MATLAB, these statements are `for`, `while`, `switch`, and `if`. In a long function, an `end` statement may be some distance away from the corresponding control statement. Even with your best efforts at organizing the code, there may be some confusion as to what `end` ends. For local clarity, indentation and comments often help:

```
function z = zap(v);
   z = [];
   for x = v;
      if x > 0;
         y = x;
         while y > 0;
            y = f(y);
            switch y;
               case 1;    y = f1(y);
               case 5;    y = f5(y);
               otherwise; y = -1;
            end % switch
         end % while
         z = [z,y];
      end % if
   end % for
```

11.16 The "break" statement

The `break` statement transfers control to the statement following the `end` of the most deeply-nested `for`-loop or `while`-loop that the `break` statement is in. However, `if` statements and `switch` statements also have `end`'s, which can confuse matters:

```
for i=1:n;
   while L;
      switch E
         Case 'A'
         if L2;
            break;   % A break statement here ...
         end; % if            /
      end; % switch          /
   end; % while             /
   % <--- ... causes transfer to here.
end;
```

If a `break` statement occurs in a `.m` file outside of any `for` or `while` loop, it acts as a `return` statement.

11.17 Functions and files

You have coded a function named "mufunc" which begins:

```
function y = mufunc(x)
```

but you have stored it in a file named "myfunc.m". In MATLAB, you must now refer to this function as "myfunc" at all times. The name "mufunc" in the function statement is ignored. If you invoke "mufunc", which, say, happens to be the name of a built-in function that you are trying to override, MATLAB will not know about your version, and may proceed with the buil-in version without error or warning. The file name is the only means MATLAB has for accessing a user-defined function. This restriction also means that you must store each of your externally-accessible functions in a separate file. In MATLAB 5, you can also have subfunctions that reside in the same file, below their primary function, but these are only accessible by the primary function.

Chapter 12

Desiderata

This chapter describes features that are not currently in MATLAB, but that the author, his students, and his clients have wished for at times. Admittedly, some of these ideas are rather crude in their present form, but with a little refining, they might become practical. They would certainly eliminate much of the tricky stuff mentioned in the previous chapter. The chapter ends with non-desideratum: the GoTo feature, which is not currently a part of MATLAB, should never be part of MATLAB, but that keeps coming up in conversation.

12.1 Statements *versus* lines of code

A MATLAB statement should end with a semicolon, and should take as many lines as the author pleases without ellipses. MATLAB should ignore the end-of-line. Printing of results should be requested explicitly, e.g. by a statement qualifier like `p:` in `p:a=b*c;`. This is far more civil than the current practice of spewing useless, and often voluminous, output whenever a semicolon is accidentally omitted. Comments should be delimited at both ends, e.g. `(* Comment *)`, and should be permitted anywhere that a blank is permitted. A comment, like a statement, could extend over several lines.

12.2 Functions *versus* files

Currently, each accessible function resides in a separate file. While there may be several subfunctions in the same file, only the top level function is accessible to the user. If the name of the function and the name of the file disagree, the file name takes precedence.

It would be preferable to divorce the concepts of file and function. A function should be defined as any code that begins with a `function` statement, and ends with `end`, whether that code came from a file or directly from the user. Currently, a user can define a function directly using the `inline` function, but only if the function consists of one statement, and has only one output argument.

Like a function, a script should begin with a `script` statement consisting of the word `script` and a name, and it should end with `end`. A file should hold as many top-level functions and scripts as the user cares to put there. Subfunctions should be placed within top-level functions and scripts. In this way, functions and scripts associated with a given process could be grouped in one file without cluttering the file directory with needless entries.

The `help` entry for a file would depend on the number of functions and scripts and their names. If a file contained exactly one top-level function or script, whose name was the same as the file, the `help` command would operate as it does now. Otherwise, The `help` command would present a list of contents with one-line descriptions. More detailed help could be obtained by the command e.g.:

```
help filename:functionname
```

12.3 Functions *versus* variables

Functions are sometimes confused with variables. Parts of MATLAB help to avoid the confusion, e.g. `feval`, while other parts seem to abet it, e.g. `exist`. The confusion can be eliminated completely as follows:

1. Array indices are enclosed in parentheses as in `x(i)`, while cell-content indices are enclosed in braces as in `c{i}`. In like manner, function arguments should be enclosed in square brackets as in `log[x]`, as multiple output arguments are enclosed now.

2. Function and script invocations should always have brackets, even with no arguments, as in `clear[]`.

3. Special built-in values like `eps`, `i`, `j`, and `pi` should be referred to as functions, e.g. `i[]`. Thus they would no longer interfere with the values or the existence status of user-defined variables with the same names.

4. With the preceding reforms, the `feval` feature could be dropped.

12.4 Functions *versus* commands

Commands like `clear`, `save`, and `load` accept open arguments, i.e arguments without parentheses. Confusingly, ordinary functions like `sin` and `log` can do likewise, as illustrated in the previous chapter, usually with odd results. Also, it seems pointless to allow statements like `log d` while disallowing `2*log d` or even `2*(log d)`. Functions should require enclosed arguments. Command/function duality seems a dubious idea. Maybe commands should have the option to enclose their arguments, although there seems no compelling reason for such an option. (The manual cites the ability to construct command arguments from character strings, but MATLAB already has the ability to construct entire commands from strings using the `eval` function.) In contrast, ordinary functions with open arguments have no merit at all.

12.5 The "fix" attribute

Indexing a target (e.g.) can produce a multitude of unwanted results: wrong sizes, data types, shapes, values, and dimensionality. There should be the option to hold specified properties of an array fixed, so that any unwanted changes are announced as errors. The pair of statements:

```
fix A;
unfix A;
```

would control the process. For example, the statement:

```
fix A   size [3,5]   type 'real';
```

would guarantee that the array `A` remain a real array of fixed size until an `unfix A` statement occurs, or until the scope of the process ends (e.g. an exit from the function where the fix occured). Any attempt to change the size of `A` or store complex numbers in `A` would raise an error.

12.6 Special values

There should be at least one special value which, when encountered in an expression, stops the proceedings and returns to the user with an error message. This value, denoted `Wrong`, may be stored but not otherwise used. The purpose of, say, `x(i)=Wrong` is to guard against further use of `x(i)` until it has been properly initialized. This value should be distinct from `NaN` and `Inf`, values which may allow the computation to proceed at great expense and no benefit whatever. Also, `NaN` and `Inf` can be produced by computation, whereas `Wrong`, when met in an expression, would serve as a clear indicator of some initialization problem.

In addition to `Wrong`, there should be logical values:

```
NaNArith  InfArith  ComplexArith
```

providing the option to disallow arithmetic in the master session or in any user-written function for which `NaN`'s, `Inf`'s, and/or complex quantities are not legitimate. For example, `ComplexArith=0` disallows computations with complex quantities within a given workspace, precluding the likes of `sqrt` and `log` with indadvertently negative arguments. At present, computations with `NaN` and `Inf` can be prevented, but only in debugging mode.

Currently, the functions `max` and `min` treat the value `NaN` as a missing value to be ignored. However, `NaN` can also be produced by erroneous computation like `0/0`, which will then be interpreted wrongly as a missing value, thus concealing a bug. There should be a special value called `Missing`, reserved for missing values in statistical operations. Unlike `NaN`, the value `Missing` could be created only by direct definition, not by errant arithmetic.

12.7 Empty structures

An array can be made empty by use of the empty symbol `[]`. When this is done by indexing the target, certain properties of the target are remembered, namely, the class of the target, all dimensions of the target except the one that was emptied, and in the case of structures, the field names are also remembered. This convention enables one to append and delete cross-sections of an array in unpredictable fashion, yet still retain consistent syntax even when the array becomes empty. This convention should be extended to structures in the case when the fields themselves are being appended and deleted, which is not the case now. For example, if array `A` becomes empty because the last structure field is removed by the `rmfield` function:

```
A = rmfield( A, 'f' )   % f (A's only field) is removed.
```

the variable `A` is now an empty `double` array, not a structure. The empty array `A` cannot be treated as a structure, i.e., fields cannot be appended to it, despite the fact that it was generated by the structure function `rmfield`. The fields of a structure are analogous to array dimensions, as evinced by MATLAB's `cell2struct` and `struct2cell` functions, and should be treated in similar fashion. The function `rmfield` should always produce structures, and if it removes the only field from a structure array, it should produce a structure array of the same size with no fields.

12.8 Indexed expressions

At present, it is illegal to index a matrix expression, e.g., as in:

```
Z = (X\Y)(I,J);
% which is equivalent to the three statements:
W = X\Y;   Z = W(I,J);   clear W;
```

The syntax in the first line should be legal. Indices following a right parenthesis in this manner have no conflicting meaning. Granted, there are times when, with sufficient cleverness, you can generate only the relevant part of a matrix. But there are many examples, like that above, where generating a larger matrix, extracting the desired part, and discarding the rest is the most efficient thing to do.

12.9 Embedded assignments

It is illegal to store intermediate results "on the fly", e.g., as in:

```
Z = A * ( Y = B+C );   % which is equivalent to ...
                       %    Y = B+C;   Z = A*Y;
```

The value produced by the embedded assignment `(Y=B+C)` is simply the value of `Y`. The embedded assignment creates no conflict with other syntax. MATLAB already differentiates the assignment symbol "`=`" from the logical equality symbol "`==`". Thus, the embedded assignment `Y=B+C` cannot be mistaken (by MATLAB) for the logical comparison `Y==B+C`.

When used together, embedded assignments and indexed intermediates can produce more concerted and more efficient code:

```
z = ( (X=(A\B)(I,J)) * (Y=(C\D)(I,J)) )(1,1);
% which replaces the seven statements:
% R = A\B;   X = R(I,J);
% R = C\D;   Y = R(I,J);
% R = X*Y;   z = R(1,1);   clear R;
```

12.10 Embedded Conditionals

The word "`if`" appears exclusively at the beginning of an `if` statement. In other languages, it has also been used to return a value, much as a function would do, e.g.:

```
Z = Z+W + (U+V)*(if q, A; else B)*(X+Y);
% which replaces the following:
% if q, Z = Z+W + (U+V)*A*(X+Y);
% else  Z = Z+W + (U+V)*B*(X+Y); end;
```

The embedded `if` can eliminate the choice between coding duplicate expressions or creating useless intermediate variables.

12.11 Output selector function

There should be a function to select the nth output argument of a function that produces multiple outputs, enabling us to use such a function in an expression no matter which output we use. For example, when using `svd`, if we want to use the third output in an expression without storing it explicitly:

```
B = A * argout(svd(X),3);
```

12.12 Bits

There are multidimensional arrays of numbers, characters, etc. In similar fashion, there should be arrays of bits. That is, the status of bits should be on a par with the status of numbers, strings, and other data classes. The current convention of manipulating bits within a mantissa of a floating point integer makes the syntax for bits considerably messier than the syntax for numbers and characters.

12.13 The find function

At present, the `find` function does nothing useful for n-dimensional arrays where $n > 2$. The extension of `find` to higher dimensions is obvious, and should be implemented.

12.14 The linspace and logspace functions

The function `linspace` is simplicity itself. The expression:

```
linspace(a,b,n)
```

produces a linearly-spaced row vector running from exact scalar `a` to exact scalar `b` with exactly `n` elements. One exception is `n=1` when $a \neq b$ (the result is `b`), which makes it impossible to satisfy all of the above conditions. The other exception is `n=0`, which is interpreted as `n=1` when it should really produce a null. Also, the linspace routine should be extended in the obvious way to handle conformal array arguments `a` and `b`, producing an array with one more index.

The goal of the function `logspace` is, presumably, to provide a similar service for log spacing. However, the function `logspace` now requires as input the *logs* of the first and last elements of the output, thus introducing the possibility of roundoff error in the end values. Further, and most bizarre, if the value `pi` is an end point, it must be entered directly, without taking its log, thus introducing a discontinuous requirement where there need not be any. The `logspace` function should behave as much like `linspace` as possible, i.e. the arguments should be first value, last value, and number of values. Of course, for log spacing, the two limits should have the same sign.

12.15 global and clear

In the last chapter, the section "global" explains some tricky stuff about the interaction of the `global` and `clear` commands. Much confusion can be eliminated by two reforms:

1. The statement `clear x` should clear global `x` from both local and global workspaces, obsoleting the form `clear global x`.
2. Global statements should create symbolic entries rather than variables with empty values, so that one could have undefined, yet global, variables.

The global statement can be made much safer than it is now. The global statement should accept an optional label, e.g.:

```
global   'L1' x y   'L2' z
```

so that a global variables `x` and `y` would be shared only with functions that used those variables with the global label 'L1', and global variable z would be shared only with functions that used that variable with global label 'L2'. Labeling global variables vastly reduces the probability that those variables would be used by other functions in a conflicting manner.

12.16 The "end" and "break" statements

To clarify code, and to enable MATLAB to give more precise diagnostics, the `end` statement should be replaced by distinct statements, e.g.:

```
endfor  endwhile  endif  endswitch endtry
```

which end the various control syntaxes in the obviuous way (see below).

The `break` statement is very restrictive, often forcing a transfer to a spot in the code where one does not wish to go. To provide more options, the `break` statement should take an integer argument as in `break(n)`. (For ease of compilation and use, the argument might be required to be an explicit integer rather than a variable.) The argument tells which relevant `end`-type statement to go to, as follows:

```
for ...
   while ...
      if ...
         switch ...
         case 1
            break(n);  % break with argument.
            ...
         endswitch;
         % <-- n=1 gets here (a redundant break).
         try ...   % The break is not in
         catch ... %   this 'try' group, so
         endtry;   %   this end doesn't count.
      endif;
      % <-- n=2 gets here.
   endwhile;
   % <-- n=3 gets here
endfor;
% <-- n=4 gets here.
```

Unlike the current `break`, the proposed `break` can transfer to the end of any `for`, `while`, `if`, or `switch` group in which the `break` resides.

12.17 The rename command

An array may have an unwieldy name, particularly if it has been loaded from a file whose name is some kind of extended mnemonic. Another case where name change is desirable is a utility script file that alters an array, and does so without forcing other copies of the array to be generated, as a function call would do. Rather than change the script code for each input array, one would prefer to change the name of the array to conform to the script. Currently, changing the name of an array involves creating a copy of it, as in:

```
A = Unwieldy_Name;   clear Unwieldy_Name;
```

which can take considerable time if the array is large. Instead, the command

```
rename Unwieldy_Name A;
```

would alter only the symbol tables, leaving the array undisturbed,

12.18 The ugly "GoTo"

This section describes a non-desideratum, one that several of my students (especially FORTRAN users) have inquired about. MATLAB, thankfully, does not allow statements of the form `GoTo L`, where `L` is a statement label. Such devices would enable us to write the following kind of code:

```
function f = factorial(n)
% Compute the factorial of a scalar non-negative integer.
          if max(size(n)) ~= 1, GoTo ErrExit;
          else GoTo NegCheck; end;
ErrExit:  error('Bad argument in factorial');
Good_n:   f = max([1,n]);
Decr_n:   n = n-1;
          if n < 2, GoTo GoodExit;
          else f = f*n;   GoTo Decr_n;   end;
Intcheck: if rem(n,1), GoTo ErrExit;
          else GoTo Good_n;   end;
NegCheck: If n < 0, GoTo ErrExit;
          else GoTo IntCheck;   end;
GoodExit: ; % Null final statement for normal exit.
```

Granted, skilled programmers would not write in such an involuted style even if they were *required* to use `GoTo` statements. Alas, programming style is not a universal gift. The abuses abetted by `GoTo` are legendary, even when the programmer is brilliant in some other field. Anything that can be done with `GoTo` can be done without it, so let's steer clear of it entirely.

Chapter 13

Overview of the help files

13.1 Help is available

MATLAB is a language with many features, some of which are easily understood, idiot-proof, and flawlessly implemented. But many features of MATLAB may lack one or another of those assets. Where bugs are concerned, whatever their source, MATLAB provides a powerful suite of debug commands, most starting with the letters "db". (Yes, even what *you* code is easier to debug in MATLAB!) Debug commands may be listed by the commands `help debug` or `help general`. If the bug you find stems from a faulty MATLAB feature, the bug can be reported to MathWorks, Inc. by email at bugs@mathworks.com, which in the author's experience, elicits a prompt and usually helpful response. Other points of communication at MathWorks, Inc. are listed on the back of the title page of any MATLAB manual. Your first line of defense against misunderstanding some feature, or missing it completely, is the help files. However, MATLAB has grown as a language, to the point that there are now thousands of help entries. If you hope to grasp what's out there, you must first read introductory material such as "Using MATLAB" from MathWorks, and "MATLAB Class Notes" from me. This will give you a sense of the language constructs and styles of data available, and where the functions of various categories fit in the scheme of things. It also helps to peruse the major help categories which are listed in this chapter.

13.2 Using this listing

At the top of each help page is the symbol >>, which is the MATLAB prompt. Following >> is what you type to get a listing of that page. Type `help` if you want a list of broad topics in the help files. Type `help elfun` for a list of elementary functions like `sqrt`. The features are listed under appropriate categories, with a brief explanation of each feature. For an expanded explanation of a particular feature, type e.g. `help sqrt`. This will give you specific information about how the feature works, including required input and the form of the output. Note especially the "See also" listings at the end of many such entries, which provide essential cross references. For example, the last line of the help entry "ops" reads "See also ARITH, RELOP, SLASH". To see material in the ARITH (arithmetic) category, type `help arith`, and similarly for the RELOP (relational operators) and SLASH (scalar and matrix division) categories. If, after your best efforts, you find the documentation unclear, erroneous, incomplete, missing, or impossible to locate, send email to doc@mathworks.com.

13.3 A case in point: the max function

Sometimes, a feature appears so obvious that help seems unnecessary. For example, consider `max`. In some languages, notably FORTRAN, the inputs to max are two numbers (scalars), and the output is the larger of the two. Never assume that MATLAB's functions will operate like those in other languages. As a case in point, here is MATLAB's response to `help max`:

```
MAX    Largest component.
For vectors, MAX(X) is the largest element in X. For matrices,
MAX(X) is a row vector containing the maximum element from each
column. For N-D arrays, MAX(X) operates along the first
non-singleton dimension.

[Y,I] = MAX(X) returns the indices of the maximum values in vector I.
If the values along the first non-singleton dimension contain more
than one maximal element, the index of the first one is returned.

MAX(X,Y) returns an array the same size as X and Y with the
largest elements taken from X or Y.  Either one can be a scalar.

[Y,I] = MAX(X,[],DIM) operates along the dimension DIM.

When complex, the magnitude MAX(ABS(X)) is used.  NaN's are
ignored when computing the maximum.

Example: If X = [2 8 4   then max(X,[],1) is [7 8 9],
                 7 3 9]

     max(X,[],2) is [8    and max(X,5) is [5 8 5
                     9],                  7 5 9].
```

From this help entry, we see that `max` can indeed take the maximum of two scalars, but in addition, a single call to `max` can produce:

- an array of pairwise maxima from two equally-sized arrays,
- an array of pairwise maxima from an array and a scalar,
- the maximum of a vector,
- maxima of a set of vectors defined by a matrix or an array, along with a specified or default dimension.
- indices of maxima within those vectors and arrays,
- proper answers in the presence of missing data (NaN's).

The type of information you get depends on how many input arguments you provide, and how many output arrays you request. MATLAB's `max` goes far beyond FORTRAN's `max`, and you need to know how. **Moral:** If a feature is new to you, use help.

13.4 Entry level help

```
>> help   (* Entry level probe into the Help files *)
(* Note: comments, marked by (* *), are for this listing only,
   and do not appear in the actual help files. *)

HELP topics:         Area of interest

matlab/general    -  General purpose commands.
matlab/ops        -  Operators and special characters.
matlab/lang       -  Programming language constructs.
matlab/elmat      -  Matrices and matrix manipulation.
matlab/elfun      -  Elementary math functions.
matlab/specfun    -  Specialized math functions.
matlab/matfun     -  Matrix algebra.
matlab/datafun    -  Data analysis, Fourier transforms.
matlab/polyfun    -  Interpolation and polynomials.
matlab/funfun     -  Function functions and ODE solvers.
matlab/sparfun    -  Sparse matrices.
matlab/graph2d    -  Two dimensional graphs.
matlab/graph3d    -  Three dimensional graphs.
matlab/specgraph  -  Specialized graphs.
matlab/graphics   -  Handle Graphics.
matlab/uitools    -  Graphical user interface tools.
matlab/strfun     -  Character strings.
matlab/iofun      -  File input/output.
matlab/timefun    -  Time and dates.
matlab/datatypes  -  Data types and structures.
matlab/demos      -  Examples and demonstrations.

For more help on directory/topic, type "help topic".

(* You can and should write "help" entries for your own code.
   See the chapter "M-files" under "Self-documentation" *)
```

13.5 General purpose commands

```
   >> help general
   General purpose commands.
   MATLAB Toolbox   Version 5.1 04-Apr-1997
General information. ------------------------------------------
  help        - On-line help, display text at command line.
  helpwin     - On-line help, separate window for navigation.
  helpdesk    - Hypertext documentation and troubleshooting.
  demo        - Run demonstrations.
  ver         - MATLAB, SIMULINK, and toolbox version information.
  whatsnew    - Display Readme files.
  Readme      - What's new in MATLAB 5.1
Managing the workspace. ---------------------------------------
  who         - List current variables.
  whos        - List current variables, long form.
  clear       - Clear variables and functions from memory.
  pack        - Consolidate workspace memory.
  load        - Load workspace variables from disk.
  save        - Save workspace variables to disk.
  quit        - Quit MATLAB session.
Managing commands and functions. -----------------------------------
  what        - List MATLAB-specific files in directory.
  type        - List M-file.
  edit        - Edit M-file.
  lookfor     - Search all M-files for keyword.
  which       - Locate functions and files.
  pcode       - Create pre-parsed pseudo-code file (P-file).
  inmem       - List functions in memory.
  mex         - Compile MEX-function.
Managing the search path. -------------------------------------
  path        - Get/set search path.
  addpath     - Add directory to search path.
  rmpath      - Remove directory from search path.
  editpath    - Modify search path.
Controlling the command window. ------------------------------------
  echo        - Echo commands in M-files.
  more        - Control paged output in command window.
  diary       - Save text of MATLAB session.
  format      - Set output format.
Operating system commands. -----------------------------------------
  cd          - Change current working directory.
  pwd         - Show (print) current working directory.
  dir         - List directory.
  delete      - Delete file.
  getenv      - Get environment variable.
  !           - Execute operating system command (see PUNCT).
  dos         - Execute DOS command and return result.
```

```
  unix        - Execute UNIX command and return result.
  vms         - Execute VMS DCL command and return result.
  web         - Open Web browser on site or files.
  computer    - Computer type.
Debugging M-files. ----------------------------------------------
  debug       - List debugging commands.
  dbstop      - Set breakpoint.
  dbclear     - Remove breakpoint.
  dbcont      - Continue execution.
  dbdown      - Change local workspace context.
  dbstack     - Display function call stack.
  dbstatus    - List all breakpoints.
  dbstep      - Execute one or more lines.
  dbtype      - List M-file with line numbers.
  dbup        - Change local workspace context.
  dbquit      - Quit debug mode.
  dbmex       - Debug MEX-files (UNIX only).
Profiling M-files. ----------------------------------------------
  profile     - Profile M-file execution time.
See also PUNCT.
```

13.6 Operators and special characters

```
>> help ops
  Operators and special characters.
  Arithmetic operators. -------------------------------------------
    plus       - Plus                            +
    uplus      - Unary plus                      +
    minus      - Minus                           -
    uminus     - Unary minus                     -
    mtimes     - Matrix multiply                 *
    times      - Array multiply                 .*
    mpower     - Matrix power                    ^
    power      - Array power                    .^
    mldivide   - Backslash or left matrix divide  \
    mrdivide   - Slash or right matrix divide     /
    ldivide    - Left array divide              .\
    rdivide    - Right array divide             ./
    kron       - Kronecker tensor product      kron
  Relational operators. -------------------------------------------
    eq         - Equal                          ==
    ne         - Not equal                      ~=
    lt         - Less than                       <
    gt         - Greater than                    >
    le         - Less than or equal             <=
    ge         - Greater than or equal          >=
  Logical operators. ----------------------------------------------
```

```
  and        - Logical AND                          &
  or         - Logical OR                           |
  not        - Logical NOT                          ~
  xor        - Logical EXCLUSIVE OR
  any        - True if any element of vector is nonzero
  all        - True if all elements of vector are nonzero
Special characters. ------------------------------------------------
  colon      - Colon                                :
  paren      - Parentheses and subscripting       ( )
  paren      - Parentheses and subscripting       ( )
  paren      - Brackets                           [ ]
  paren      - Braces and subscripting            { }
  paren      - Braces and subscripting            { }
  punct      - Decimal point                       .
  punct      - Structure field access              .
  punct      - Parent directory                    ..
  punct      - Continuation                        ...
  punct      - Separator                           ,
  punct      - Semicolon                           ;
  punct      - Comment                             %
  punct      - Invoke operating system command     !
  punct      - Assignment                          =
  punct      - Quote                               '
  transpose  - Transpose                          .'
  ctranspose - Complex conjugate transpose         '
  horzcat    - Horizontal concatenation           [,]
  vertcat    - Vertical concatenation             [;]
  subsasgn   - Subscripted assignment          ( ),{ },.
  subsref    - Subscripted reference           ( ),{ },.
  subsindex  - Subscript index
Bitwise operators. -------------------------------------------------
  bitand     - Bit-wise AND.
  bitcmp     - Complement bits.
  bitor      - Bit-wise OR.
  bitmax     - Maximum floating point integer.
  bitxor     - Bit-wise XOR.
  bitset     - Set bit.
  bitget     - Get bit.
  bitshift   - Bit-wise shift.
Set operators. -----------------------------------------------------
  union      - Set union.
  unique     - Set unique.
  intersect  - Set intersection.
  setdiff    - Set difference.
  setxor     - Set exclusive-or.
  ismember   - True for set member.
See also ARITH, RELOP, SLASH.
```

13.7 Programming language constructs

```
>> help lang
  Programming language constructs.
  Control flow. ------------------------------------------------------
    if          - Conditionally execute statements.
    else        - IF statement condition.
    elseif      - IF statement condition.
    end         - Terminate scope of FOR, WHILE, SWITCH and IF statements.
    for         - Repeat statements a specific number of times.
    while       - Repeat statements an indefinite number of times.
    break       - Terminate execution of WHILE or FOR loop.
    switch      - Switch among several cases based on expression.
    case        - SWITCH statement case.
    otherwise   - Default SWITCH statement case.
    return      - Return to invoking function.
  Evaluation and execution. ------------------------------------------
    eval        - Execute string with MATLAB expression.
    feval       - Execute function specified by string.
    evalin      - Evaluate expression in workspace.
    builtin     - Execute built-in function from overloaded method.
    assignin    - Assign variable in workspace.
    run         - Run script.
  Scripts, functions, and variables. ---------------------------------
    script      - About MATLAB scripts and M-files.
    function    - Add new function.
    global      - Define global variable.
    mfilename   - Name of currently executing M-file.
    lists       - Comma separated lists.
    exist       - Check if variables or functions are defined.
    isglobal    - True for global variables.
  Argument handling. -------------------------------------------------
    nargchk     - Validate number of input arguments.
    nargin      - Number of function input arguments.
    nargout     - Number of function output arguments.
    varargin    - Variable length input argument list.
    varargout   - Variable length output argument list.
    inputname   - Input argument name.
  Message display. ---------------------------------------------------
    error       - Display error message and abort function.
    warning     - Display warning message.
    lasterr     - Last error message.
    errortrap   - Skip error during testing.
    disp        - Display an array.
    fprintf     - Display formatted message.
    sprintf     - Write formatted data to a string.
  Interactive input. -------------------------------------------------
    input       - Prompt for user input.
```

```
  keyboard    - Invoke keyboard from M-file.
  pause       - Wait for user response.
  uimenu      - Create user interface menu.
  uicontrol   - Create user interface control.
```

13.8 Matrix manipulation

```
>> help elmat
  Elementary matrices and matrix manipulation.
  Elementary matrices. ---------------------------------------------
    zeros       - Zeros array.
    ones        - Ones array.
    eye         - Identity matrix.
    repmat      - Replicate and tile array.
    rand        - Uniformly distributed random numbers.
    randn       - Normally distributed random numbers.
    linspace    - Linearly spaced vector.
    logspace    - Logarithmically spaced vector.
    meshgrid    - X and Y arrays for 3-D plots.
    :           - Regularly spaced vector and index into matrix.
  Basic array information. -----------------------------------------
    size        - Size of matrix.
    length      - Length of vector.
    ndims       - Number of dimensions.
    disp        - Display matrix or text.
    isempty     - True for empty matrix.
    isequal     - True if arrays are identical.
    isnumeric   - True for numeric arrays.
    islogical   - True for logical array.
    logical     - Convert numeric values to logical.
  Matrix manipulation. ---------------------------------------------
    reshape     - Change size.
    diag        - Diagonal matrices and diagonals of matrix.
    tril        - Extract lower triangular part.
    triu        - Extract upper triangular part.
    fliplr      - Flip matrix in left/right direction.
    flipud      - Flip matrix in up/down direction.
    flipdim     - Flip matrix along specified dimension.
    rot90       - Rotate matrix 90 degrees.
    :           - Regularly spaced vector and index into matrix.
    find        - Find indices of nonzero elements.
    end         - Last index.
    sub2ind     - Linear index from multiple subscripts.
    ind2sub     - Multiple subscripts from linear index.
  Special variables and constants. ---------------------------------
    ans         - Most recent answer.
    eps         - Floating point relative accuracy.
```

```
  realmax     - Largest positive floating point number.
  realmin     - Smallest positive floating point number.
  pi          - 3.1415926535897....
  i, j        - Imaginary unit.
  inf         - Infinity.
  NaN         - Not-a-Number.
  isnan       - True for Not-a-Number.
  isinf       - True for infinite elements.
  isfinite    - True for finite elements.
  flops       - Floating point operation count.
  why         - Succinct answer.
Specialized matrices. ------------------------------------------------
  compan      - Companion matrix.
  gallery     - Higham test matrices.
  hadamard    - Hadamard matrix.
  hankel      - Hankel matrix.
  hilb        - Hilbert matrix.
  invhilb     - Inverse Hilbert matrix.
  magic       - Magic square.
  pascal      - Pascal matrix.
  rosser      - Classic symmetric eigenvalue test problem.
  toeplitz    - Toeplitz matrix.
  vander      - Vandermonde matrix.
  wilkinson   - Wilkinson's eigenvalue test matrix.
```

13.9 Elementary math functions

```
>> help elfun
  Elementary math functions.
  Trigonometric. -------------------------------------------------------
  sin         - Sine.
  sinh        - Hyperbolic sine.
  asin        - Inverse sine.
  asinh       - Inverse hyperbolic sine.
  cos         - Cosine.
  cosh        - Hyperbolic cosine.
  acos        - Inverse cosine.
  acosh       - Inverse hyperbolic cosine.
  tan         - Tangent.
  tanh        - Hyperbolic tangent.
  atan        - Inverse tangent.
  atan2       - Four quadrant inverse tangent.
  atanh       - Inverse hyperbolic tangent.
  sec         - Secant.
  sech        - Hyperbolic secant.
  asec        - Inverse secant.
  asech       - Inverse hyperbolic secant.
```

```
  csc         - Cosecant.
  csch        - Hyperbolic cosecant.
  acsc        - Inverse cosecant.
  acsch       - Inverse hyperbolic cosecant.
  cot         - Cotangent.
  coth        - Hyperbolic cotangent.
  acot        - Inverse cotangent.
  acoth       - Inverse hyperbolic cotangent.
Exponential. ----------------------------------------------------
  exp         - Exponential.
  log         - Natural logarithm.
  log10       - Common (base 10) logarithm.
  log2        - Base 2 logarithm and dissect floating point number.
  pow2        - Base 2 power and scale floating point number.
  sqrt        - Square root.
  nextpow2    - Next higher power of 2.
Complex. --------------------------------------------------------
  abs         - Absolute value.
  angle       - Phase angle.
  conj        - Complex conjugate.
  imag        - Complex imaginary part.
  real        - Complex real part.
  unwrap      - Unwrap phase angle.
  isreal      - True for real array.
  cplxpair    - Sort numbers into complex conjugate pairs.
Rounding and remainder. -----------------------------------------
  fix         - Round towards zero.
  floor       - Round towards minus infinity.
  ceil        - Round towards plus infinity.
  round       - Round towards nearest integer.
  mod         - Modulus (signed remainder after division).
  rem         - Remainder after division.
  sign        - Signum.
```

13.10 Specialized math functions

```
>> help specfun
  Specialized math functions.
  Specialized math functions. -------------------------------------
  airy        - Airy functions.
  besselj     - Bessel function of the first kind.
  bessely     - Bessel function of the second kind.
  besselh     - Bessel functions of the third kind (Hankel function).
  besseli     - Modified Bessel function of the first kind.
  besselk     - Modified Bessel function of the second kind.
  beta        - Beta function.
  betainc     - Incomplete beta function.
```

```
    betaln      - Logarithm of beta function.
    ellipj      - Jacobi elliptic functions.
    ellipke     - Complete elliptic integral.
    erf         - Error function.
    erfc        - Complementary error function.
    erfcx       - Scaled complementary error function.
    erfinv      - Inverse error function.
    expint      - Exponential integral function.
    gamma       - Gamma function.
    gammainc    - Incomplete gamma function.
    gammaln     - Logarithm of gamma function.
    legendre    - Associated Legendre function.
    cross       - Vector cross product.
  Number theoretic functions. ----------------------------------------
    factor      - Prime factors.
    isprime     - True for prime numbers.
    primes      - Generate list of prime numbers.
    gcd         - Greatest common divisor.
    lcm         - Least common multiple.
    rat         - Rational approximation.
    rats        - Rational output.
    perms       - All possible permutations.
    nchoosek    - All combinations of N elements taken K at a time.
  Coordinate transforms. ---------------------------------------------
    cart2sph    - Transform Cartesian to spherical coordinates.
    cart2pol    - Transform Cartesian to polar coordinates.
    pol2cart    - Transform polar to Cartesian coordinates.
    sph2cart    - Transform spherical to Cartesian coordinates.
    hsv2rgb     - Convert hue-saturation-value colors to red-green-blue.
    rgb2hsv     - Convert red-green-blue colors to hue-saturation-value.
```

13.11 Numerical linear algebra

```
>> help matfun
  Matrix functions - numerical linear algebra.
  Matrix analysis. ---------------------------------------------------
    norm        - Matrix or vector norm.
    normest     - Estimate the matrix 2-norm.
    rank        - Matrix rank.
    det         - Determinant.
    trace       - Sum of diagonal elements.
    null        - Null space.
    orth        - Orthogonalization.
    rref        - Reduced row echelon form.
    subspace    - Angle between two subspaces.
  Linear equations. --------------------------------------------------
    \ and /     - Linear equation solution; use "help slash".
```

```
  inv         - Matrix inverse.
  cond        - Condition number with respect to inversion.
  condest     - 1-norm condition number estimate.
  chol        - Cholesky factorization.
  cholinc     - Incomplete Cholesky factorization.
  lu          - LU factorization.
  luinc       - Incomplete LU factorization.
  qr          - Orthogonal-triangular decomposition.
  nnls        - Non-negative least-squares.
  pinv        - Pseudoinverse.
  lscov       - Least squares with known covariance.
Eigenvalues and singular values. ---------------------------------------
  eig         - Eigenvalues and eigenvectors.
  svd         - Singular value decomposition.
  eigs        - A few eigenvalues.
  svds        - A few singular values.
  poly        - Characteristic polynomial.
  polyeig     - Polynomial eigenvalue problem.
  condeig     - Condition number with respect to eigenvalues.
  hess        - Hessenberg form.
  qz          - QZ factorization for generalized eigenvalues.
  schur       - Schur decomposition.
Matrix functions. ------------------------------------------------------
  expm        - Matrix exponential.
  logm        - Matrix logarithm.
  sqrtm       - Matrix square root.
  funm        - Evaluate general matrix function.
Factorization utilities. -----------------------------------------------
  qrdelete    - Delete column from QR factorization.
  qrinsert    - Insert column in QR factorization.
  rsf2csf     - Real block diagonal form to complex diagonal form.
  cdf2rdf     - Complex diagonal form to real block diagonal form.
  balance     - Diagonal scaling to improve eigenvalue accuracy.
  planerot    - Given's plane rotation.
```

13.12 Data analysis and Fourier transforms

```
>> help datafun
  Data analysis and Fourier transforms.
  Basic operations. ------------------------------------------------------
  max         - Largest component.
  min         - Smallest component.
  mean        - Average or mean value.
  median      - Median value.
  std         - Standard deviation.
  sort        - Sort in ascending order.
  sortrows    - Sort rows in ascending order.
```

```
  sum         - Sum of elements.
  prod        - Product of elements.
  hist        - Histogram.
  trapz       - Trapezoidal numerical integration.
  cumsum      - Cumulative sum of elements.
  cumprod     - Cumulative product of elements.
  cumtrapz    - Cumulative trapezoidal numerical integration.
Finite differences. -------------------------------------------------
  diff        - Difference and approximate derivative.
  gradient    - Approximate gradient.
  del2        - Discrete Laplacian.
Correlation. ---------------------------------------------------------
  corrcoef    - Correlation coefficients.
  cov         - Covariance matrix.
  subspace    - Angle between subspaces.
Filtering and convolution. -------------------------------------------
  filter      - One-dimensional digital filter.
  filter2     - Two-dimensional digital filter.
  conv        - Convolution and polynomial multiplication.
  conv2       - Two-dimensional convolution.
  convn       - N-dimensional convolution.
  deconv      - Deconvolution and polynomial division.
Fourier transforms. --------------------------------------------------
  fft         - Discrete Fourier transform.
  fft2        - Two-dimensional discrete Fourier transform.
  fftn        - N-dimensional discrete Fourier Transform.
  ifft        - Inverse discrete Fourier transform.
  ifft2       - Two-dimensional inverse discrete Fourier transform.
  ifftn       - N-dimensional inverse discrete Fourier Transform.
  fftshift    - Shift DC component to center of spectrum.
Sound and audio. -----------------------------------------------------
  sound       - Play vector as sound.
  soundsc     - Autoscale and play vector as sound.
  speak       - Convert input string to speech (Macintosh only).
  recordsound - Record sound (Macintosh only).
  soundcap    - Sound capabilities (Macintosh only).
  mu2lin      - Convert mu-law encoding to linear signal.
  lin2mu      - Convert linear signal to mu-law encoding.
Audio file inport/export. --------------------------------------------
  auwrite     - Write NeXT/SUN (".au") sound file.
  auread      - Read NeXT/SUN (".au") sound file.
  wavwrite    - Write Microsoft WAVE (".wav") sound file.
  wavread     - Read Microsoft WAVE (".wav") sound file.
  readsnd     - Read SND resources and files (Macintosh only).
  writesnd    - Write SND resources and files (Macintosh only)
```

13.13 Interpolation and polynomials

```
>> help polyfun
  Interpolation and polynomials.
  Data interpolation. ------------------------------------------------
    interp1     - 1-D interpolation (table lookup).
    interp1q    - Quick 1-D linear interpolation.
    interpft    - 1-D interpolation using FFT method.
    interp2     - 2-D interpolation (table lookup).
    interp3     - 3-D interpolation (table lookup).
    interpn     - N-D interpolation (table lookup).
    griddata    - Data gridding and surface fitting.
  Spline interpolation. ----------------------------------------------
    spline      - Cubic spline interpolation.
    ppval       - Evaluate piecewise polynomial.
  Geometric analysis. ------------------------------------------------
    delaunay    - Delaunay triangulation.
    dsearch     - Search Delaunay triagulation for nearest point.
    tsearch     - Closest triangle search.
    convhull    - Convex hull.
    voronoi     - Voronoi diagram.
    inpolygon   - True for points inside polygonal region.
    rectint     - Rectangle intersection area.
    polyarea    - Area of polygon.
  Polynomials. -------------------------------------------------------
    roots       - Find polynomial roots.
    poly        - Convert roots to polynomial.
    polyval     - Evaluate polynomial.
    polyvalm    - Evaluate polynomial with matrix argument.
    residue     - Partial-fraction expansion (residues).
    polyfit     - Fit polynomial to data.
    polyder     - Differentiate polynomial.
    conv        - Multiply polynomials.
    deconv      - Divide polynomials.
```

13.14 Functions of functions, and ODE solvers

```
>> help funfun
  Function functions and ODE solvers.
  Optimization and root finding. -------------------------------------
    fmin     - Minimize function of one variable.
    fmins    - Minimize function of several variables.
    fzero    - Find zero of function of one variable.
  Numerical integration (quadrature). --------------------------------
    quad     - Numerically evaluate integral, low order method.
    quad8    - Numerically evaluate integral, higher order method.
    dblquad  - Numerically evaluate double integral.
```

```
  Plotting. ------------------------------------------------------------
    ezplot    - Easy to use function plotter.
    fplot     - Plot function.
  Inline function object. ----------------------------------------------
    inline    - Construct INLINE function object.
    argnames  - Argument names.
    formula   - Function formula.
    char      - Convert INLINE object to character array.
  Utilities. -----------------------------------------------------------
    vectorize - Vectorize string expression or INLINE function object.
  Ordinary differential equation solvers. ------------------------------
  (If unsure about stiffness, try ODE45 first, then ODE15S.)
    ode45     - Solve non-stiff differential equations, medium order method.
    ode23     - Solve non-stiff differential equations, low order method.
    ode113    - Solve non-stiff differential equations, variable order method.
    ode15s    - Solve stiff differential equations, variable order method.
    ode23s    - Solve stiff differential equations, low order method.
    odefile   - ODE file syntax.
  ODE Option handling. -------------------------------------------------
    odeset    - Create/alter ODE OPTIONS structure.
    odeget    - Get ODE OPTIONS parameters.
  ODE output functions. ------------------------------------------------
    odeplot   - Time series ODE output function.
    odephas2  - 2-D phase plane ODE output function.
    odephas3  - 3-D phase plane ODE output function.
    odeprint  - Command window printing ODE output function.
```

13.15 Sparse matrices

```
>> help sparfun
  Sparse matrices.
  Elementary sparse matrices. ------------------------------------------
    speye       - Sparse identity matrix.
    sprand      - Sparse uniformly distributed random matrix.
    sprandn     - Sparse normally distributed random matrix.
    sprandsym   - Sparse random symmetric matrix.
    spdiags     - Sparse matrix formed from diagonals.
  Full to sparse conversion. -------------------------------------------
    sparse      - Create sparse matrix.
    full        - Convert sparse matrix to full matrix.
    find        - Find indices of nonzero elements.
    spconvert   - Import from sparse matrix external format.
  Working with sparse matrices. ----------------------------------------
    nnz         - Number of nonzero matrix elements.
    nonzeros    - Nonzero matrix elements.
    nzmax       - Amount of storage allocated for nonzero matrix elements.
    spones      - Replace nonzero sparse matrix elements with ones.
```

```
  spalloc     - Allocate space for sparse matrix.
  issparse    - True for sparse matrix.
  spfun       - Apply function to nonzero matrix elements.
  spy         - Visualize sparsity pattern.
Reordering algorithms. ----------------------------------------------
  colmmd      - Column minimum degree permutation.
  symmmd      - Symmetric minimum degree permutation.
  symrcm      - Symmetric reverse Cuthill-McKee permutation.
  colperm     - Column permutation.
  randperm    - Random permutation.
  dmperm      - Dulmage-Mendelsohn permutation.
Linear algebra. -----------------------------------------------------
  eigs        - A few eigenvalues.
  svds        - A few singular values.
  luinc       - Incomplete LU factorization.
  cholinc     - Incomplete Cholesky factorization.
  normest     - Estimate the matrix 2-norm.
  condest     - 1-norm condition number estimate.
  sprank      - Structural rank.
Linear Equations (iterative methods). -------------------------------
  pcg         - Preconditioned Conjugate Gradients Method.
  bicg        - BiConjugate Gradients Method.
  bicgstab    - BiConjugate Gradients Stabilized Method.
  cgs         - Conjugate Gradients Squared Method.
  gmres       - Generalized Minimum Residual Method.
  qmr         - Quasi-Minimal Residual Method.
Operations on graphs (trees). ---------------------------------------
  treelayout  - Lay out tree or forest.
  treeplot    - Plot picture of tree.
  etree       - Elimination tree.
  etreeplot   - Plot elimination tree.
  gplot       - Plot graph, as in "graph theory".
Miscellaneous. ------------------------------------------------------
  symbfact    - Symbolic factorization analysis.
  spparms     - Set parameters for sparse matrix routines.
  spaugment   - Form least squares augmented system.
```

13.16 Two-dimensional graphics

```
>> help graph2d
  Two dimensional graphs.
  Elementary X-Y graphs. ----------------------------------------------
    plot      - Linear plot.
    loglog    - Log-log scale plot.
    semilogx  - Semi-log scale plot.
    semilogy  - Semi-log scale plot.
    polar     - Polar coordinate plot.
```

```
  plotyy    - Graphs with y tick labels on the left and right.
Axis control. ------------------------------------------------------
  axis      - Control axis scaling and appearance.
  zoom      - Zoom in and out on a 2-D plot.
  grid      - Grid lines.
  box       - Axis box.
  hold      - Hold current graph.
  axes      - Create axes in arbitrary positions.
  subplot   - Create axes in tiled positions.
Graph annotation. --------------------------------------------------
  legend    - Graph legend.
  title     - Graph title.
  xlabel    - X-axis label.
  ylabel    - Y-axis label.
  text      - Text annotation.
  gtext     - Place text with mouse.
Hardcopy and printing. ---------------------------------------------
  print     - Print graph or SIMULINK system; or save graph to M-file.
  printopt  - Printer defaults.
  orient    - Set paper orientation.
See also GRAPH3D, SPECGRAPH.
```

13.17 Three-dimensional graphics

```
>> help graph3d
  Three dimensional graphs.
  Elementary 3-D plots. ----------------------------------------------
    plot3     - Plot lines and points in 3-D space.
    mesh      - 3-D mesh surface.
    surf      - 3-D colored surface.
    fill3     - Filled 3-D polygons.
  Color control. -----------------------------------------------------
    colormap  - Color look-up table.
    caxis     - Pseudocolor axis scaling.
    shading   - Color shading mode.
    hidden    - Mesh hidden line removal mode.
    brighten  - Brighten or darken color map.
  Lighting. ----------------------------------------------------------
    surfl     - 3-D shaded surface with lighting.
    lighting  - Lighting mode.
    material  - Material reflectance mode.
    specular  - Specular reflectance.
    diffuse   - Diffuse reflectance.
    surfnorm  - Surface normals.
  Color maps. --------------------------------------------------------
    hsv       - Hue-saturation-value color map.
    hot       - Black-red-yellow-white color map.
```

```
    gray       - Linear gray-scale color map.
    bone       - Gray-scale with tinge of blue color map.
    copper     - Linear copper-tone color map.
    pink       - Pastel shades of pink color map.
    white      - All white color map.
    flag       - Alternating red, white, blue, and black color map.
    lines      - Color map with the line colors.
    colorcube  - Enhanced color-cube color map.
    jet        - Variant of HSV.
    prism      - Prism color map.
    cool       - Shades of cyan and magenta color map.
    autumn     - Shades of red and yellow color map.
    spring     - Shades of magenta and yellow color map.
    winter     - Shades of blue and green color map.
    summer     - Shades of green and yellow color map.
  Axis control. ------------------------------------------------------
    axis       - Control axis scaling and appearance.
    zoom       - Zoom in and out on a 2-D plot.
    grid       - Grid lines.
    box        - Axis box.
    hold       - Hold current graph.
    axes       - Create axes in arbitrary positions.
    subplot    - Create axes in tiled positions.
  Viewpoint control. -------------------------------------------------
    view       - 3-D graph viewpoint specification.
    viewmtx    - View transformation matrix.
    rotate3d   - Interactively rotate view of 3-D plot.
  Graph annotation. --------------------------------------------------
    title      - Graph title.
    xlabel     - X-axis label.
    ylabel     - Y-axis label.
    zlabel     - Z-axis label.
    colorbar   - Display color bar (color scale).
    text       - Text annotation.
    gtext      - Mouse placement of text.
  Hardcopy and printing. ---------------------------------------------
    print      - Print graph or SIMULINK system; or save graph to M-file.
    printopt   - Printer defaults.
    orient     - Set paper orientation.
  See also GRAPH2D, SPECGRAPH.
```

13.18 Specialized graphics

```
>> help specgraph
  Specialized graphs.
  Specialized 2-D graphs. --------------------------------------------
    area       - Filled area plot.
```

```
  bar         - Bar graph.
  barh        - Horizontal bar graph.
  bar3        - 3-D bar graph.
  bar3h       - Horizontal 3-D bar graph.
  comet       - Comet-like trajectory.
  errorbar    - Error bar plot.
  ezplot      - Easy to use function plotter.
  feather     - Feather plot.
  fill        - Filled 2-D polygons.
  fplot       - Plot function.
  hist        - Histogram.
  pareto      - Pareto chart.
  pie         - Pie chart.
  pie3        - 3-D pie chart.
  plotmatrix - Scatter plot matrix.
  ribbon      - Draw 2-D lines as ribbons in 3-D.
  stem        - Discrete sequence or "stem" plot.
  stairs      - Stairstep plot.
Contour and 2-1/2 D graphs. ----------------------------------------
  contour     - Contour plot.
  contourf    - Filled contour plot.
  contour3    - 3-D Contour plot.
  clabel      - Contour plot elevation labels.
  pcolor      - Pseudocolor (checkerboard) plot.
  quiver      - Quiver plot.
  voronoi     - Voronoi diagram.
Specialized 3-D graphs. --------------------------------------------
  comet3      - 3-D comet-like trajectories.
  meshc       - Combination mesh/contour plot.
  meshz       - 3-D mesh with curtain.
  stem3       - 3-D stem plot.
  quiver3     - 3-D quiver plot.
  slice       - Volumetric slice plot.
  surfc       - Combination surf/contour plot.
  trisurf     - Triangular surface plot.
  trimesh     - Triangular mesh plot.
  waterfall   - Waterfall plot.
Images display and file I/O. ---------------------------------------
  image       - Display image.
  imagesc     - Scale data and display as image.
  colormap    - Color look-up table.
  gray        - Linear gray-scale color map.
  contrast    - Gray scale color map to enhance image contrast.
  brighten    - Brighten or darken color map.
  colorbar    - Display color bar (color scale).
  imread      - Read image from graphics file.
  imwrite     - Write image to graphics file.
```

```
  imfinfo    - Information about graphics file.
Movies and animation. ------------------------------------------------
  capture    - Screen capture of current figure.
  moviein    - Initialize movie frame memory.
  getframe   - Get movie frame.
  movie      - Play recorded movie frames.
  qtwrite    - Translate movie into QuickTime format (Macintosh only).
  rotate     - Rotate object about specified orgin and direction.
  frame2im   - Convert movie frame to indexed image.
  im2frame   - Convert index image into movie format.
Color related functions. ---------------------------------------------
  spinmap    - Spin color map.
  rgbplot    - Plot color map.
  colstyle   - Parse color and style from string.
Solid modeling. ------------------------------------------------------
  cylinder   - Generate cylinder.
  sphere     - Generate sphere.
  patch      - Create patch.
See also GRAPH2D, GRAPH3D.
```

13.19 Handle graphics

```
>> help graphics
  Handle Graphics.
  Figure window creation and control. ----------------------------------
    figure     - Create figure window.
    gcf        - Get handle to current figure.
    clf        - Clear current figure.
    shg        - Show graph window.
    close      - Close figure.
    refresh    - Refresh figure.
  Axis creation and control. -------------------------------------------
    subplot    - Create axes in tiled positions.
    axes       - Create axes in arbitrary positions.
    gca        - Get handle to current axes.
    cla        - Clear current axes.
    axis       - Control axis scaling and appearance.
    box        - Axis box.
    caxis      - Control pseudocolor axis scaling.
    hold       - Hold current graph.
    ishold     - Return hold state.
  Handle Graphics objects. ---------------------------------------------
    figure     - Create figure window.
    axes       - Create axes.
    line       - Create line.
    text       - Create text.
    patch      - Create patch.
```

```
    surface    - Create surface.
    image      - Create image.
    light      - Create light.
    uicontrol  - Create user interface control.
    uimenu     - Create user interface menu.
  Handle Graphics operations. ----------------------------------------
    set        - Set object properties.
    get        - Get object properties.
    reset      - Reset object properties.
    delete     - Delete object.
    gco        - Get handle to current object.
    gcbo       - Get handle to current callback object.
    gcbf       - Get handle to current callback figure.
    drawnow    - Flush pending graphics events.
    findobj    - Find objects with specified property values.
    copyobj    - Make copy of graphics object and its children.
  Hardcopy and printing. ---------------------------------------------
    print      - Print graph or SIMULINK system; or save graph to M-file.
    printopt   - Printer defaults.
    orient     - Set paper orientation.
  Utilities. ---------------------------------------------------------
    closereq   - Figure close request function.
    newplot    - M-file preamble for NextPlot property.
    ishandle   - True for graphics handles.
  See also GRAPH2D, GRAPH3D, SPECGRAPH.
```

13.20 Graphical user interface

```
>> help uitools
  Graphical user interface tools.
  GUI functions. -----------------------------------------------------
    uicontrol  - Create user interface control.
    uimenu     - Create user interface menu.
    ginput     - Graphical input from mouse.
    dragrect   - Drag XOR rectangles with mouse.
    rbbox      - Rubberband box.
    selectmoveresize   - Interactively select, move, resize, or copy objects.
    waitforbuttonpress - Wait for key/buttonpress over figure.
    waitfor    - Block execution and wait for event.
    uiwait     - Block execution and wait for resume.
    uiresume   - Resume execution of blocked M-file.
  GUI design tools. --------------------------------------------------
    guide       - Design GUI.
    align       - Align uicontrols and axes.
    cbedit      - Edit callback.
    menuedit    - Edit menu.
    propedit    - Edit property.
```

```
Dialog boxes. ------------------------------------------------------------
  dialog     - Create dialog figure.
  axlimdlg   - Axes limits dialog box.
  errordlg   - Error dialog box.
  helpdlg    - Help dialog box.
  inputdlg   - Input dialog box.
  listdlg    - List selection dialog box.
  menu       - Generate menu of choices for user input.
  msgbox     - Message box.
  questdlg   - Question dialog box.
  warndlg    - Warning dialog box.
  uigetfile  - Standard open file dialog box.
  uiputfile  - Standard save file dialog box.
  uisetcolor - Color selection dialog box.
  uisetfont  - Font selection dialog box.
  pagedlg    - Page position dialog box.
  printdlg   - Print dialog box.
  waitbar    - Display wait bar.
Menu utilities. ----------------------------------------------------------
  makemenu   - Create menu structure.
  menubar    - Computer dependent default setting for MenuBar property.
  umtoggle   - Toggle "checked" status of uimenu object.
  winmenu    - Create submenu for "Window" menu item.
Toolbar button group utilities. ------------------------------------------
  btngroup   - Create toolbar button group.
  btnstate   - Query state of toolbar button group.
  btnpress   - Button press manager for toolbar button group.
  btndown    - Depress button in toolbar button group.
  btnup      - Raise button in toolbar button group.
User-defined figure/axes property utilities. -----------------------------
  clruprop   - Clear user-defined property.
  getuprop   - Get value of user-defined property.
  setuprop   - Set user-defined property.
Miscellaneous utilities. -------------------------------------------------
  allchild   - Get all object children.
  findall    - Find all objects.
  hidegui    - Hide/unhide GUI.
  edtext     - Interactive editing of axes text objects.
  getstatus  - Get status text string in figure.
  setstatus  - Set status text string in figure.
  popupstr   - Get popup menu selection string.
  remapfig   - Transform figure objects' positions.
  setptr     - Set figure pointer.
  getptr     - Get figure pointer.
  overobj    - Get handle of object the pointer is over.
```

13.21 Character strings

```
>> help strfun
  Character strings.
  General. ------------------------------------------------------------
    char        - Create character array (string).
    double      - Convert string to numeric character codes.
    cellstr     - Create cell array of strings from character array.
    blanks      - String of blanks.
    deblank     - Remove trailing blanks.
    eval        - Execute string with MATLAB expression.
  String tests. -------------------------------------------------------
    ischar      - True for character array (string).
    iscellstr   - True for cell array of strings.
    isletter    - True for letters of the alphabet.
    isspace     - True for white space characters.
  String operations. --------------------------------------------------
    strcat      - Concatenate strings.
    strvcat     - Vertically concatenate strings.
    strcmp      - Compare strings.
    strncmp     - Compare first N characters of strings.
    findstr     - Find one string within another.
    strjust     - Justify character array.
    strmatch    - Find possible matches for string.
    strrep      - Replace string with another.
    strtok      - Find token in string.
    upper       - Convert string to uppercase.
    lower       - Convert string to lowercase.
  String to number conversion. ----------------------------------------
    num2str     - Convert number to string.
    int2str     - Convert integer to string.
    mat2str     - Convert matrix to eval'able string.
    str2num     - Convert string to number.
    sprintf     - Write formatted data to string.
    sscanf      - Read string under format control.
  Base number conversion. ---------------------------------------------
    hex2num     - Convert IEEE hexadecimal to double precision number.
    hex2dec     - Convert hexadecimal string to decimal integer.
    dec2hex     - Convert decimal integer to hexadecimal string.
    bin2dec     - Convert binary string to decimal integer.
    dec2bin     - Convert decimal integer to binary string.
    base2dec    - Convert base B string to decimal integer.
    dec2base    - Convert decimal integer to base B string.
  See also STRINGS.
```

13.22 Input and output

```
>> help iofun
  File input/output.
  File opening and closing. ------------------------------------------
    fopen       - Open file.
    fclose      - Close file.
  Binary file I/O. ---------------------------------------------------
    fread       - Read binary data from file.
    fwrite      - Write binary data to file.
  Formatted file I/O. ------------------------------------------------
    fscanf      - Read formatted data from file.
    fprintf     - Write formatted data to file.
    fgetl       - Read line from file, discard newline character.
    fgets       - Read line from file, keep newline character.
    input       - Prompt for user input.
  String conversion. -------------------------------------------------
    sprintf     - Write formatted data to string.
    sscanf      - Read string under format control.
  File positioning. --------------------------------------------------
    ferror      - Inquire file error status.
    feof        - Test for end-of-file.
    fseek       - Set file position indicator.
    ftell       - Get file position indicator.
    frewind     - Rewind file.
  File name handling. ------------------------------------------------
    matlabroot  - Root directory of MATLAB installation.
    filesep     - Directory separator for this platform.
    pathsep     - Path separator for this platform.
    mexext      - MEX filename extension for this platform.
    fullfile    - Build full filename from parts.
    partialpath - Partial pathnames.
    tempdir     - Get temporary directory.
    tempname    - Get temporary file.
  File import/export functions. --------------------------------------
    load        - Load workspace from MAT-file.
    save        - Save workspace to MAT-file.
    dlmread     - Read ASCII delimited file.
    dlmwrite    - Write ASCII delimited file.
    wk1read     - Read spreadsheet (WK1) file.
    wk1write    - Write spreadsheet (WK1) file.
  Image file import/export. ------------------------------------------
    imread      - Read image from graphics file.
    imwrite     - Write image to graphics file.
    imfinfo     - Return information about graphics file.
  Audio file import/export. ------------------------------------------
    auwrite     - Write NeXT/SUN (".au") sound file.
    auread      - Read NeXT/SUN (".au") sound file.
```

```
    wavwrite    - Write Microsoft WAVE (".wav") sound file.
    wavread     - Read Microsoft WAVE (".wav") sound file.
  Command window I/O -------------------------------------------------
    clc         - Clear command window.
    home        - Send cursor home.
    disp        - Display array.
    input       - Prompt for user input.
    pause       - Wait for user response.
```

13.23 Time and dates

```
>> help timefun
  Time and dates.
  Current date and time. ---------------------------------------------
    now         - Current date and time as date number.
    date        - Current date as date string.
    clock       - Current date and time as date vector.
  Basic functions. ---------------------------------------------------
    datenum     - Serial date number.
    datestr     - String representation of date.
    datevec     - Date components.
  Date functions. ----------------------------------------------------
    calendar    - Calendar.
    weekday     - Day of week.
    eomday      - End of month.
    datetick    - Date formatted tick labels.
  Timing functions. --------------------------------------------------
    cputime     - CPU time in seconds.
    tic, toc    - Stopwatch timer.
    etime       - Elapsed time.
    pause       - Wait in seconds.
```

13.24 Data types and structures

```
>> help datatypes
  Data types and structures.
  Data types (classes). ----------------------------------------------
    double      - Convert to double precision.
    sparse      - Create sparse matrix.
    char        - Create character array (string).
    cell        - Create cell array.
    struct      - Create or convert to structure array.
    uint8       - Convert to unsigned 8-bit integer.
    inline      - Construct INLINE object.
  Multi-dimensional array functions. ---------------------------------
    cat         - Concatenate arrays.
```

```
  ndims       - Number of dimensions.
  ndgrid      - Generate arrays for N-D functions and interpolation.
  permute     - Permute array dimensions.
  ipermute    - Inverse permute array dimensions.
  shiftdim    - Shift dimensions.
  squeeze     - Remove singleton dimensions.
Cell array functions. ------------------------------------------------
  cell        - Create cell array.
  celldisp    - Display cell array contents.
  cellplot    - Display graphical depiction of cell array.
  num2cell    - Convert numeric array into cell array.
  deal        - Deal inputs to outputs.
  cell2struct - Convert cell array into structure array.
  struct2cell - Convert structure array into cell array.
  iscell      - True for cell array.
Structure functions. -------------------------------------------------
  struct      - Create or convert to structure array.
  fieldnames  - Get structure field names.
  getfield    - Get structure field contents.
  setfield    - Set structure field contents.
  rmfield     - Remove structure field.
  isfield     - True if field is in structure array.
  isstruct    - True for structures.
Object oriented programming functions. -------------------------------
  class       - Create object or return object class.
  struct      - Convert object to structure array.
  methods     - Display class method names.
  isa         - True if object is a given class.
  isobject    - True for objects.
  inferiorto  - Inferior class relationship.
  superiorto  - Superior class relationship.
Overloadable operators. ----------------------------------------------
  minus       - Overloadable method for a-b.
  plus        - Overloadable method for a+b.
  times       - Overloadable method for a.*b.
  mtimes      - Overloadable method for a*b.
  mldivide    - Overloadable method for a\b.
  mrdivide    - Overloadable method for a/b.
  rdivide     - Overloadable method for a./b.
  ldivide     - Overloadable method for a.\b.
  power       - Overloadable method for a.^b.
  mpower      - Overloadable method for a^b.
  uminus      - Overloadable method for -a.
  uplus       - Overloadable method for +a.
  horzcat     - Overloadable method for [a b].
  vertcat     - Overloadable method for [a;b].
  le          - Overloadable method for a<=b.
```

```
  lt          - Overloadable method for a<b.
  gt          - Overloadable method for a>b.
  ge          - Overloadable method for a>=b.
  eq          - Overloadable method for a==b.
  ne          - Overloadable method for a~=b.
  not         - Overloadable method for ~a.
  and         - Overloadable method for a&b.
  or          - Overloadable method for a|b.
  subsasgn    - Overloadable method for a(i)=b, a{i}=b, and a.field=b.
  subsref     - Overloadable method for a(i), a{i}, and a.field.
  colon       - Overloadable method for a:b.
  transpose   - Overloadable method for a.'
  ctranspose  - Overloadable method for a'
  subsindex   - Overloadable method for x(a).
```

13.25 Examples and demonstrations

```
>> help demos
  Examples and demonstrations. ---------------------------------------------
    Type 'demo' at the command line to browse more demos of
    MATLAB, the Toolboxes, and SIMULINK.
  MATLAB/Introduction. -----------------------------------------------------
    demo        - Browse demos for MATLAB, Toolboxes, and SIMULINK
  MATLAB/Matrices. ---------------------------------------------------------
    intro       - Introduction to basic matrix operations in MATLAB.
    inverter    - Demonstrate the inversion of a matrix.
    buckydem    - Connectivity graph of the Buckminster Fuller geodesic dome.
    sparsity    - Demonstrate effect of sparsity orderings.
    matmanip    - Introduction to matrix manipulation.
    eigmovie    - Symmetric eigenvalue movie.
    rrefmovie   - Computation of Reduced Row Echelon Form.
    delsqdemo   - Finite difference Laplacian on various domains.
    sepdemo     - Separators for a finite element mesh.
    airfoil     - Display sparse matrix from NASA airfoil.
  MATLAB/Numerics. ---------------------------------------------------------
    funfuns     - Demonstrate functions that operate on other functions.
    fitdemo     - Nonlinear curve fit with simplex algorithm.
    sunspots    - FFT: the answer is 11.08, what is the question?
    e2pi        - 2D visual solutions: Which is greater, e^pi or pi^e?
    bench       - MATLAB Benchmark.
    fftdemo     - Use of the fast finite Fourier transform.
    census      - Try to predict the US population in the year 2000.
    spline2d    - Demonstrate GINPUT and SPLINE in two dimensions.
    lotkademo   - An example of ordinary differential equation solution.
    quaddemo    - Adaptive quadrature.
    zerodemo    - Zerofinding with fzero.
    fplotdemo   - Plot a function.
```

```
  quake       - Loma Prieta Earthquake.
MATLAB/Visualization. ------------------------------------------------
  graf2d      - 2D Plots: Demonstrate XY plots in MATLAB.
  graf2d2     - 3D Plots: Demonstrate XYZ plots in MATLAB.
  grafcplx    - Demonstrate complex function plots in MATLAB.
  lorenz      - Plot the orbit around the Lorenz chaotic attractor.
  imageext    - Image colormaps: changing and rotating colormaps.
  xpklein     - Klein bottle demo.
  vibes       - Vibration movie: Vibrating L-shaped membrane.
  xpsound     - Visualizing sound: Demonstrate MATLAB's sound capability.
  imagedemo   - Demonstrate MATLAB's image capability.
  penny       - Several views of the penny data.
  earthmap    - View Earth's topography.
  xfourier    - Graphic demo of Fourier series expansion.
  colormenu   - Select color map.
  cplxdemo    - Maps of functions of a complex variable.
MATLAB/Language. -----------------------------------------------------
  xplang      - Introduction to the MATLAB language.
  hndlgraf    - Demonstrate Handle Graphics for line plots.
  graf3d      - Demonstrate Handle Graphics for surface plots.
  hndlaxis    - Demonstrate Handle Graphics for axes.
MATLAB/ODE Suite. ----------------------------------------------------
  odedemo     - Demo for the ODE suite integrators.
  a2ode       - Stiff problem, linear with real eigenvalues (A2 of EHL).
  a3ode       - Stiff problem, linear with real eigenvalues (A3 of EHL).
  b5ode       - Stiff problem, linear with complex eigenvalues (B5 of EHL).
  ballode     - Equations of motion for a bouncing ball used by BALLDEMO.
  besslode    - Bessel's equation of order 0 used by BESSLDEMO.
  brussode    - Stiff problem modelling a chemical reaction (Brusselator).
  buiode      - Stiff problem with analytical solution due to Bui.
  chm6ode     - Stiff problem CHM6 from Enright and Hull.
  chm7ode     - Stiff problem CHM7 from Enright and Hull.
  chm9ode     - Stiff problem CHM9 from Enright and Hull.
  d1ode       - Stiff problem, nonlinear with real eigenvalues (D1 of EHL).
  fem1ode     - Stiff problem with a time-dependent mass matrix.
  fem2ode     - Stiff problem with a time-independent mass matrix.
  gearode     - Stiff problem due to Gear as quoted by van der Houwen.
  hb1ode      - Stiff problem 1 of Hindmarsh and Byrne.
  hb2ode      - Stiff problem 2 of Hindmarsh and Byrne.
  hb3ode      - Stiff problem 3 of Hindmarsh and Byrne.
  orbitode    - Restricted 3 body problem used by ORBITDEMO.
  orbt2ode    - Non-stiff problem D5 of Hull et al.
  rigidode    - Euler equations of a rigid body without external forces.
  sticode     - A spring-driven mass stuck to surface, used by STICDEMO.
  vdpode      - Parameterizable van der Pol equation (stiff for large mu).
Extras/Gallery. ------------------------------------------------------
  knot        - Tube surrounding a three-dimensional knot.
```

```
  quivdemo    - Demonstrate the quiver function.
  klein1      - Construct a Klein bottle.
  cruller     - Construct cruller.
  tori4       - Hoops: Construct four linked tori.
  spharm2     - Construct spherical surface harmonic.
  modes       - Plot 12 modes of the L-shaped membrane.
  logo        - Display the MATLAB L-shaped membrane logo.
Extras/Games. -------------------------------------------------------------
  xpbombs     - Minesweeper game.
  life        - Conway's Game of Life.
  bblwrap     - Bubblewrap.
  soma        - Soma cube.
Extras/Miscellaneous. -----------------------------------------------------
  truss       - Animation of a bending bridge truss.
  travel      - Traveling salesman problem.
  spinner     - Colorful lines spinning through space.
  xpquad      - Superquadrics plotting demonstration.
  codec       - Alphabet transposition coder/decoder.
  xphide      - Visual perception of objects in motion.
  makevase    - Generate and plot a surface of revolution.
  wrldtrv     - Great circle flight routes around the globe.
  logospin    - Movie of The MathWorks' logo spinning.
  crulspin    - Spinning cruller movie.
  quatdemo    - Quaternion rotation.
General Demo/Helper functions. --------------------------------------------
  cmdlnwin    - An Demo gateway routine for playing command line demos.
  cmdlnbgn    - Set up for command line demos.
  cmdlnend    - clean up after command line demos.
  finddemo    - Finds demos available for individual toolboxes.
  helpfun     - Utility function for displaying help text conveniently.
  pltmat      - Display a matrix in a figure window.
MATLAB/Helper functions. --------------------------------------------------
  bucky       - The graph of the Buckminster Fuller geodesic dome.
  peaks       - A sample function of two variables.
  membrane    - Generate MathWorks's logo.
See also SIMDEMOS
```